TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

JYRI KANNINEN

TEST AUTOMATION FOR PROCESS AUTOMATION SYSTEM

Master of Science Thesis

Examiner: Professor Matti Vilkko
Examiner and topic approved by the Council of the Faculty of Engineering Sciences on 5. November 2014

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Master's Degree Programme in Automation Technology
KANNINEN, JYRI: Test Automation for Process Automation System
Master of Science Thesis, 57 pages, 3 Appendix pages
January 2015
Major: Process automation
Examiner: Professor Matti Vilkko
Keywords: Test automation, Process Automation System, Control system, Systems engineering method, V-model

Automation system manufacturers have been developing automation application software for decades. Recently, the amount of test automation integrated in the application research and development has been on the rise due to the increase in the number of version management and continuous integration platforms developed. However, the main problem in developing test automation for automation systems is due to great differences in software architecture and interfaces between the manufacturers, which make the use of many commercial options difficult. In this thesis, the goal is to research and develop an initial test automation concept for paper machine's machine-direction control system, using MetsoDNA as the automation system platform, to decrease the work hours needed in manual testing during a development process. The systems engineering method V-model was used as a structure in the development of the test automation concept.

Initially, a workflow was identified which the developer, when developing a new product, has to go through in order the test the functionality of the new product. This workflow was seen as the single most crucial aspect that needed to be automated, in addition to the functional testing of an actual system. The test concept exploration was conducted by interviewing knowledge workers who have been involved in researching, designing and implementing test automation for automation systems in a different product scope. From the interviews, two alternatives, keyword-driven and data-driven concepts, rose as feasible concepts, which were further analyzed using a trade-off analysis. Data-driven test concept was chosen as the most feasible option, and a fully functional prototype was constructed to make a solid case for further development and to identify any major issues which might pose a development risk for the product scope of this thesis. After successful prototype evaluation, the test concept went through design phase in which the reporting from each workflow phase was implemented. The evaluation of the system was done by running the workflow multiple times with a project specification and preliminary functional tests. As a result, the developed and deployed test automation concept saves the developer from manually performing the workflow of testing a new product, which is directly relative to saved work hours in development.

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO
Automaatiotekniikan koulutusohjelma
KANNINEN, JYRI: Automaatiojärjestelmän testiautomaatio
Diplomityö, 57 sivua, 3 liitesivua
Tammikuu 2015
Pääaine: Prosessien hallinta
Tarkastaja: professori Matti Vilkko
Avainsanat: Testiautomaatio, Automaatiojärjestelmä, Säätöjärjestelmä, Systeemitekniikan metodi, V-malli

Automaatiojärjestelmien valmistajat ovat kehittäneet automaatio-ohjelmistoja jo vuosikymmeniä. Viime aikoina testiautomaation määrä ohjelmistojen tuotekehityksessä on ollut kasvussa version hallinnan ja jatkuvan integroinnin alustojen kehityksen ansiosta. Pääongelmana automaatiojärjestelmien testiautomaation kehityksessä on kuitenkin suuret erot automaatio-ohjelmistojen arkkitehtuurissa sekä rajapinnoissa, jotka tekevät useimpien kaupallisten testiautomaatio vaihtoehtojen implementoinnin vaikeaksi. Tässä työssä on tarkoituksena tutkia ja kehittää alustava testiautomaatio konsepti paperikoneen konesuuntaisen säädön säätöjärjestelmälle, joka käyttää automaatiojärjestelmänä MetsoDNA:ta. Tavoitteena on vähentää henkilötyönä tapahtuvaa testausta uuden tuotteen kehitystyön aikana. Testiautomaation kehityksessä käytettiin apuna systeemisuunnittelun metodia V-malli.

Työn aluksi tunnistettiin työketju, jonka kehittäjä käy läpi testatessaan uuden tuotteen toiminnallisuutta. Tämä työketju nähtiin tärkeimpänä yksittäisenä osiona joka tarvitsi automatisoida, todellisen järjestelmän funktionaalisen testauksen lisäksi. Mahdollisten testikonseptien tutkiminen suoritettiin haastattelemalla tietotyöläisiä jotka ovat olleet mukana kehittämässä, suunnittelemassa ja implementoimassa testiautomaatiota automaatiojärjestelmiin. Haastatteluista nousi esille kaksi mahdollista testikonseptia: Aineisto-ohjattu (data-driven) ja avainsana-ohjattu (keyword-driven). Molempien testikonseptien heikkouksia ja vahvuuksia analysoitiin, minkä perusteella aineisto-ohjattu testikonsepti valittiin sopivimmaksi vaihtoehdoksi. Aineisto-ohjatusta testikonseptista rakennettiin täysin toimiva prototyyppi, jotta konseptin jatkokehitys olisi perusteltua ja mahdolliset epäkohdat järjestelmän implementoimisessa säätöjärjestelmään saataisiin identifioitua. Onnistuneen prototyypin jälkeen implementoitiin testikonseptiin suunnitteluvaiheessa raportointi mahdollisuudet jokaisesta työketjun vaiheesta. Järjestelmän toiminnan todentaminen ja arviointi suoritettiin ajamalla työketju läpi useita kertoja eräällä projekti määrittelyllä sekä alustavalla funktionaalisella testillä. Tuloksena tuotettu testi automaatio konsepti säästää kehittäjän uuden tuotteen henkilötyönä tapahtuvalta testaukselta, mikä on suoraan suhteessa säästettyjen työtuntien määrään kehitystyössä.

PREFACE

This thesis was made for Metso Automation Oy in Tampere Finland. The topic of the thesis was interesting, even though I had little knowledge of automated testing beforehand. The topic utilized knowledge from my majors; Process automation and Systems analysis.

First I would like to thank the people at Metso Automation Oyj who made it possible for me to conduct my thesis: Niko Posti, Mira Kivimäki and Jarmo Ollanketo. Many thanks also for Jarmo in instructing and providing the tools for me during the thesis. I would also like to thank the interviewees from the R&D department who provided me with plenty of invaluable information which was crucial for the success of this thesis; Mika Karaila, Seppo Virtanen and Perttu Kotiluoto. From the Tampere University of Technology I would like to thank Professor Matti Vilkko for helping with the writing process and keeping the structure of the thesis logical.

Tampere 21.1.2014

Jyri Kanninen

TABLE OF CONTENTS

TERMS AND DEFINITIONS

MetsoDNA	Automation software platform developed by Metso Oyj.
IT	Information technology.
MOE	Measure of effectiveness.
CONOPS	Concept of operations.
SUT	System Under Test.
R&D	Research and Development.
PLC	Programmable Logical Controller.
ROI	Return On Investment.
MD	Machine Direction.
PCS	Process Control Station.
IE	Internet Explorer.
UI	User-interface.
MBT	Model-Based-Testing.
FBD	Function Block Diagram.
IEC	International Electrotechnical Commission.
MD5	Message-Digest algorithm 5.
XML	Extensible Markup Language.

1. INTRODUCTION

The advances in information technology (IT) have been the driving element in information revolution, shaping the modern industry of commerce, finance, education and entertainment. IT engineers have given rise to a wide range of software-controlled systems which are embedded in our every-day appliances.

This chapter presents the concepts of software development from systems engineering viewpoint used to analyze and improve applications of automation systems. The emphasis on this thesis is on the applications of the automation systems. The potential problems that might arise in a software project are discussed. Also, the latest innovations in the field of software development are revised and research questions with their goals and methods are presented. After reading this chapter the reader should be aware of the purpose of this thesis.

1.1 Software engineering

Software is the means by which a modern digital computer can be directed to refine multiple sources of data into a useful action. Software engineering may be defined as the systematic design and development of software products and the management of the software process [1]. Recently, software engineering has converged with systems engineering, creating a new area of expertise called software systems engineering. Systems engineering is the design, production and maintenance of trustworthy systems within cost and time constraints [2, p. 15]. The term software systems engineering refers to the application of principles to the software engineering disciple [3, p. 356]. The responsibilities in each field of expertise in a software engineering process are depicted in Figure 1.

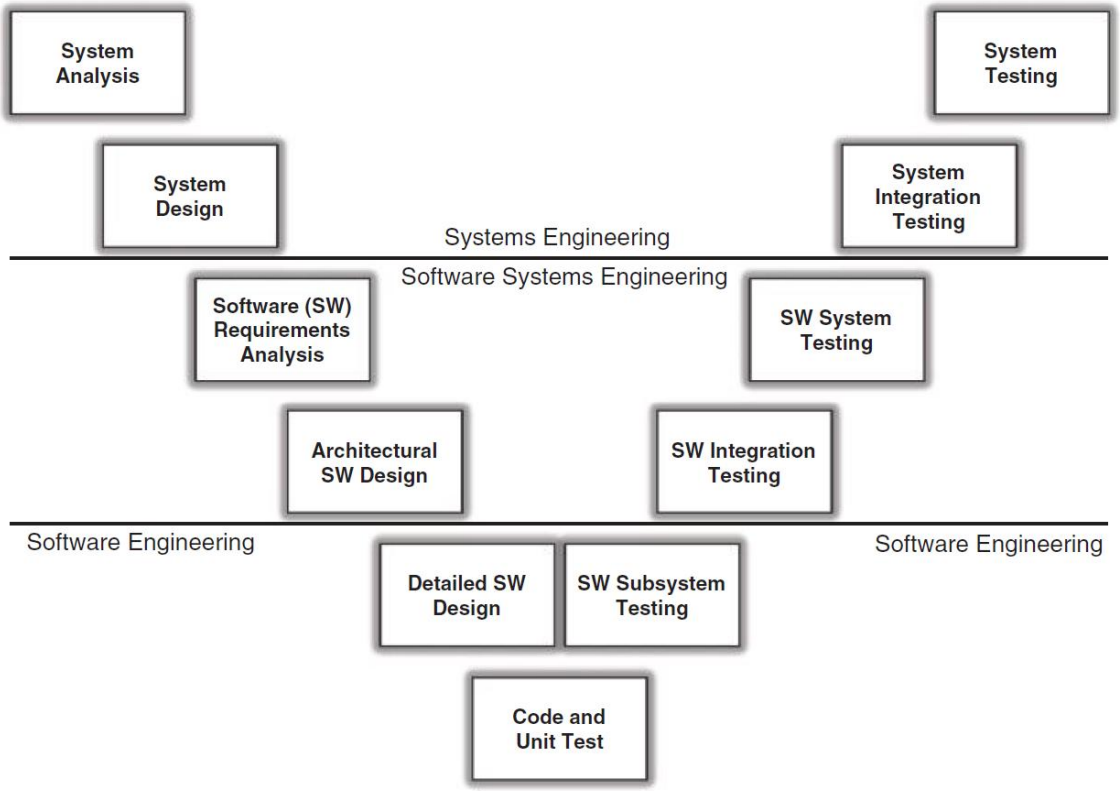

Figure 1. A software systems engineering process depicted using a "Vee" diagram [3, p. 357].

The use of systems, software and software systems engineering might promote independence between the development teams. This means that, after system design, hardware and software engineers diverge and begin developing their own respective components independently which creates problems in the integration and implementation stage. These problems usually originate from software management problems and lack of educated leadership in the said fields.

The amount of errors in coding and unit test phase tends to prolong software delivery times and postpone software acceptance testing phase. This problem partly originates from a poor choice of programming language, which should be chosen to support the nature of the application. Programming language impacts characteristics of a software product such as maintainability, portability and readability. [3]

Software engineering often requires the engineer to have a different way of thinking, also known as out-of-the-box thinking. This is to create software for systems that require many subsystems to work seamlessly independently and as a whole system. It is hard to acquire such employees with said skills, since it takes experience and education to develop the necessary state of mind. Large variations in experience, visual and logical skill tend to create contradictions in design which complicates the implementation phase.

In large and complex software-based systems, the effort to make changes in software can often be falsely perceived as trivial. The impacts of merely altering a few lines of code can be very difficult to predict accurately and a small change may require retesting of the entire software. This might lead to repetitive manual routine check-ups in the development phase which can result in employees becoming frustrated and produce errors in the final software version.

1.2 State of research

Many software development projects fail when they need to change, adapt to new user requirements or the requirements change during the development project. For this reason, many studies regarding new methods for software development have been researched by academic and commercial institutions in the early 2000s [3, p. 372]. These adaptive software methodologies are referred as agile development models.

The agile development models are based on the Agile Manifesto, which was published in 2001 [4]. Agile methods use an iterative life cycle to produce new prototypes frequently and each one is evaluated by the customer or end user. The advantages of this method are in small projects where the customer's requirements cannot be strictly defined and the customer is committed in developing the software throughout the whole project. The disadvantage is that the methodology strongly relies on end user's commitment and interaction, which in a large project can prove to be a challenge. It should also be noted that agile methods require the development teams to conduct simultaneous activities such as requirement analysis and design. [3]

1.3 Research questions and goals

The main goal of this thesis is to find the means and requirements to produce test automation for automation system based applications. The research questions are listed below. The first problem is related to the designing of test automation system, how to use the systems engineering method to decrease the amount of work hours used to test automation applications during development. The second research questions is about scoping the system architecture and understanding the cause-effect relationships between application modules in test automation applications to produce a solid base for the first research question. The third question is to find out the characteristics and specifications for test automation in a specific scope of automation system applications.

> **RQ 1:** How to decrease the amount work hours used for testing during the development of automation system applications with software systems engineering?

RQ 2: What are the special characteristics of automation system applications regarding automated testing?

RQ 3: What is the structure of automation system applications and what are their mutual dependencies?

Research problem one is answered with applying the systems engineering method "V-Model" to create levels of abstractions and allocate test levels, such as integration, module and system testing. These levels and allocations are used to create a test automation application following the structure of the V-model and then comparing the result to the existing automation application. The structure of this method is depicted in figure 1.

Research problem two is answered by interview method. The employees involved in researching, designing and implementing test automation for automation applications are interviewed in an unstructured interview. The results from the interviews are analyzed and the characteristics of already implemented test automation applications are identified.

Research problem three is answered by using graph theory method that utilizes the directed digraphs. The method is used in along with various tools to research which modules and sub-products in an automation application have a cause-effect between them.

Many automated test applications have been implemented in automation systems which include testing for user-interface (UI) and module integrity [5]. For the purposes of this thesis, they included some elements and structures that can be used in defining and developing test automation for automation applications. A small scale proof-of-concept example was made, in which the test automation tests the integrity and validity of control actions.

The structure of this thesis is organized as follows: Chapter 2 will present the methods used to acquire and analyze information from different sources. In Chapter 3, the methods are applied to present a proof-of-concept test automation system. This chapter will state the amount of work hours and workflow used in development testing and discusses the need of test automation, and how the presented systems engineering methodology is used to create the test automation system and how the amount of work hours decreases in the new proposed concept. Chapter 4 will present the results from the deployed test system which answers to the research question one. Chapter 5 evaluates and discusses the results, their practicality and how the further development of the test automation concept.

2. METHODS

This chapter will discuss the different methods used to obtain information. Information about the characteristics of automated testing systems has been mostly acquired through interviews.

2.1 V-Model in systems engineering

New more sophisticated systems tend to require increasingly larger portions of resources when they mature from concepts through engineering and to operational use. If the resources are not allocated properly throughout this life cycle, the operational system might not function as intended or could even incorporate severe design flaws. These sophisticated systems also often use new technologies which involve risks that must be identified and resolved in the development process as early as possible. The cost of fixing a design flaw later in the development phase grows multifold. To decrease the possibility of these two risks from manifesting, system development should be conducted in a step-by-step fashion where the success of each taken step is shown and the validity to take the next one is presented. This step-by-step evolution, which can also be perceived as a system's life cycle, is commonly used to depict the evolution of a system from concept through development, implementation, and operation to the ultimate disposal or replacement of the system.[3]

The classical way to depict a step-by-step evolution in software engineering is with a waterfall model. The first formal description of the waterfall model was introduced in 1970 by Royce [6], even though he didn't use the term "waterfall". The term waterfall was originally introduced in 1976 by Boehm [7]. The waterfall model is rarely used in modern software engineering projects but its basic structure can be recognized in modern software engineering, even in agile methods [3]. The major problem in waterfall model is that it's impossible to perfect one specific phase in software development, which makes it impossible to move on to the next phase in waterfall model [8].

The more modern presentation of a software development process is the V-model, which is actually an extension of the waterfall model. One way to depict the V-model is shown in Figure 2. The left side of the "V", decomposition and definition, focuses on the operational needs and engineering of the system while the right side depicts the integration and qualification activities on the engineered design. The emphasis on this

model is on the activities that the engineers perform in the engineering process of a system [9].

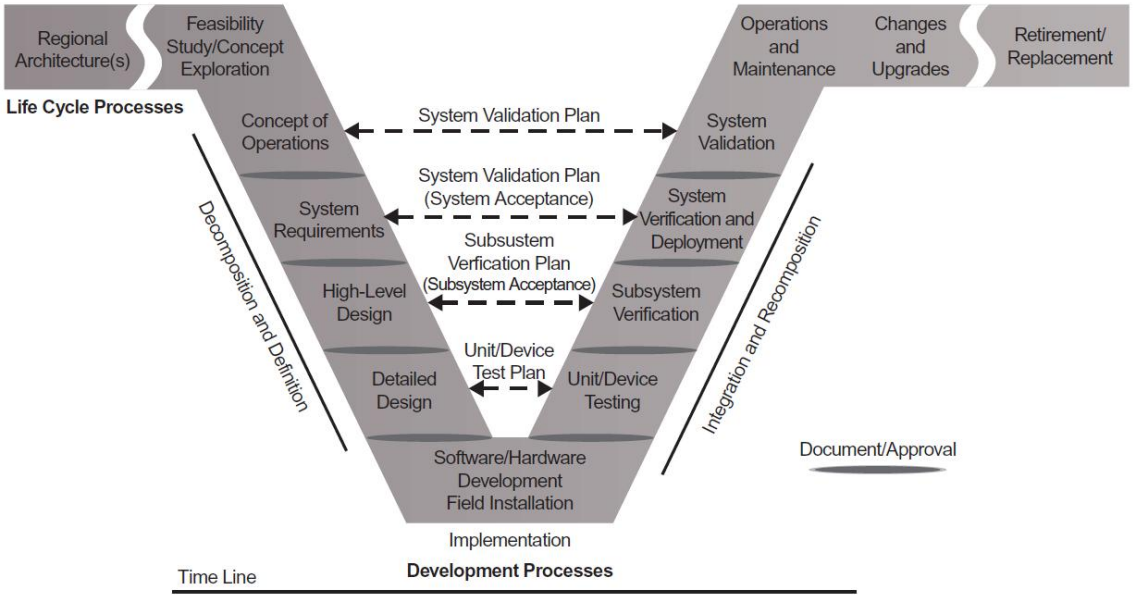

Figure 2. V-model [3, p. 36]

There are numerous ways to divide a V-model into multiple phases, for example a three-phase or twenty two-phase life cycle presented by Sage [10, pp. 32-37]. In this thesis, a structure for a concept development process using the V-model presented by Kossiakoff [3, pp. 131-314] is used as a basic design, which can roughly be divided into three main stages presented in Figure 3. The presented structure is aimed for development of large complex systems but can be as such be easily implemented into smaller projects. Some phases of the development process have been simplified to make the structure of the concept fit better for the scope if this thesis. Due to the nature of the applications produced in this thesis, the post development stage will not be used to deploy the system.

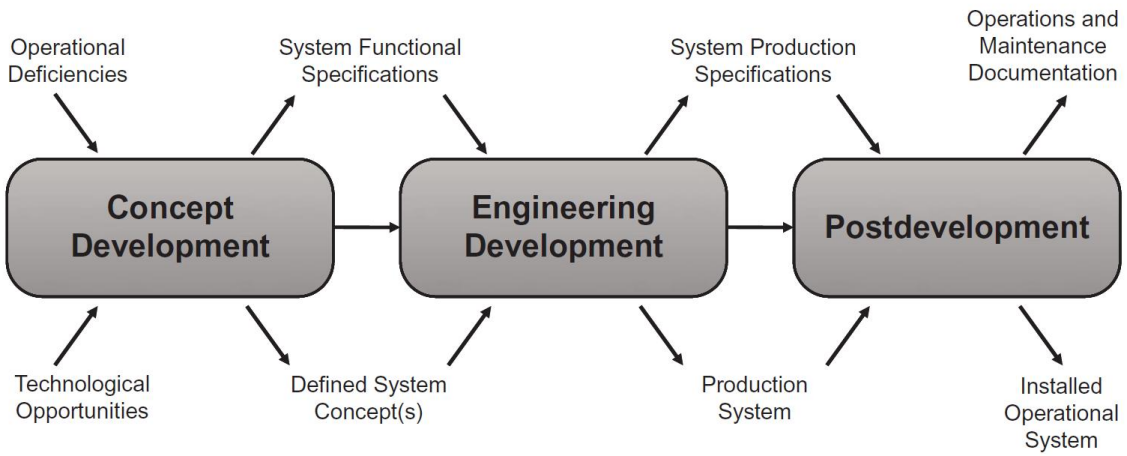

Figure 3. Three main stages of a development process [3, p. 75].

The concept development phase is the initial stage which performs the architecting of the system. It can be said that the success or failure of the project is determined in this stage by the system decisions [3]. Concept development consists of three sub-stages: Needs analysis, concept exploration and concept definition. These sub-stages are presented in Figure 4.

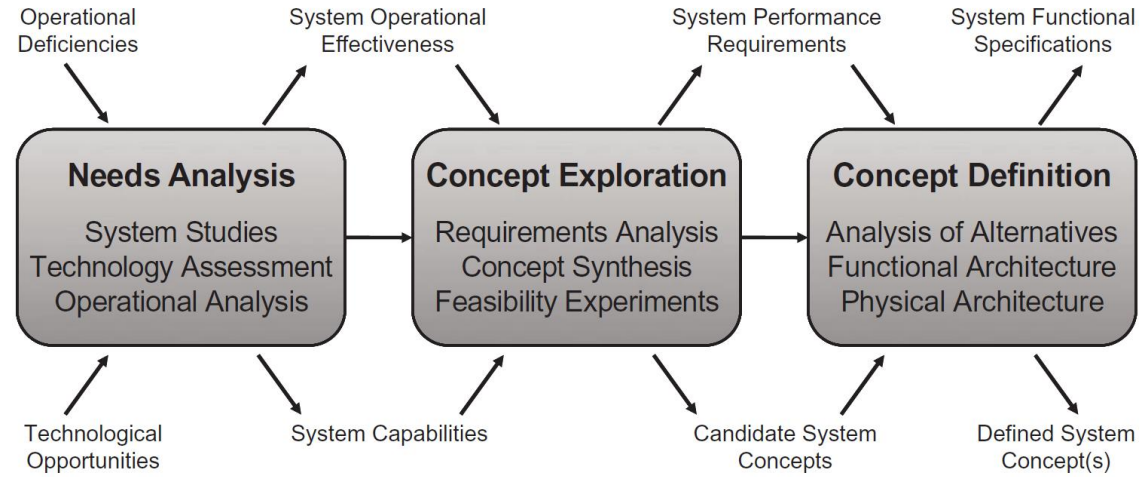

Figure 4. Sub-stages of concept development [3, p. 76].

The engineering development phase handles the implementation of chosen system concept into software and hardware components and the validation of the operational capabilities of the system. Engineering development consists of three sub-stages: Advanced development, engineering design and integration and evaluation. These sub-stages are presented in Figure 5.

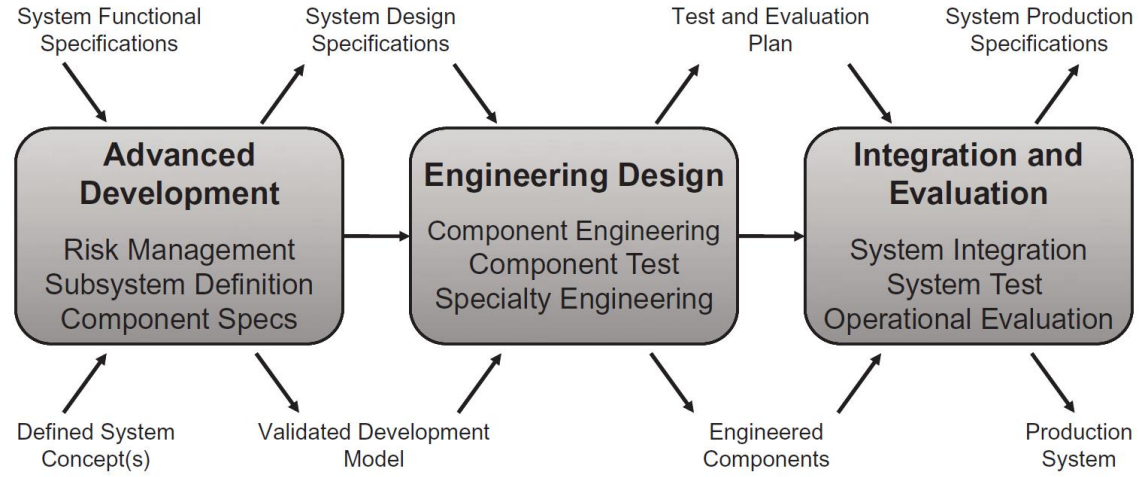

Figure 5. Sub-stages of engineering development [3, p. 78].

2.1.1 Needs analysis

The main goal for needs analysis is to show that there is a potential market and a valid need for a new system, or to carry out a major upgrade on an existing system, and to find at least one feasible approach to meet that need. Even though assessment of the market potential is a part of the needs analysis, in this thesis it's not taken into account. Needs analysis addresses the questions "Is there a valid need for a new system" and "Is

there a practical approach to satisfying such a need”. To answer these questions we must perform an analysis to produce a set of operational requirements which describe the systems purpose, capabilities and the way it is deployed. The analysis should be done multiple times in an iterative manner, but the operational requirements should be defined loosely in the first iterations, since the goal of this analysis is to demonstrate that there is at least one feasible system concept to achieve the projected need which justifies further development. Needs analysis can be divided into four main actions: Operations analysis, functional analysis, feasibility analysis and needs validation. The complete flow diagram of needs analysis presented by Kossiakoff et al. [3, p. 147] is modified such that represents the structure of this thesis and is depicted in Figure 6. [3]

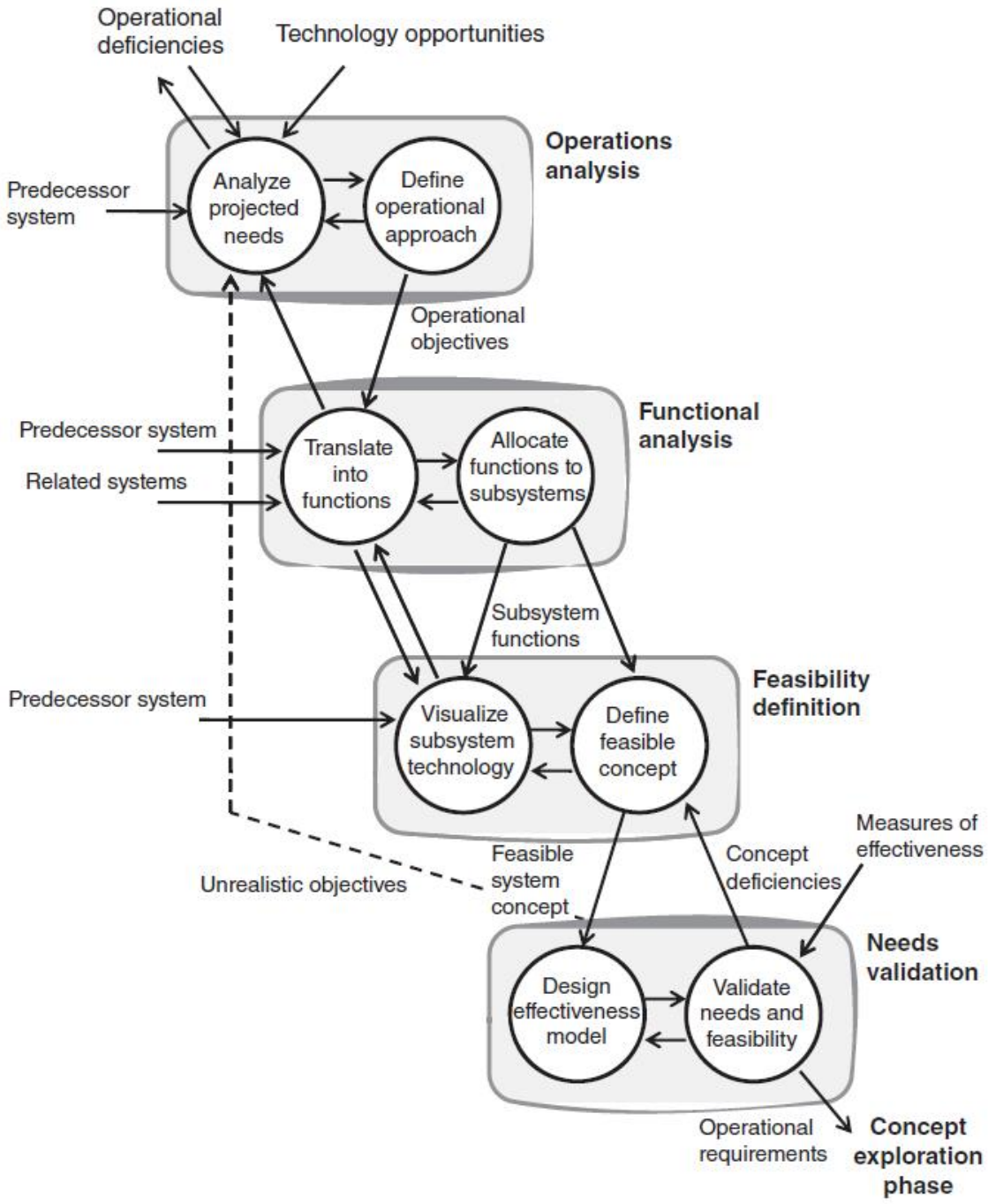

Figure 6. Needs analysis flow diagram.

Operations analysis analyzes the need for the system, which might be based on the predecessor system being deficient, system becoming obsolete or the lowered operating expenses in implementing new advantageous technology. In more customer-oriented business, it's also important to take into account customer's view on important product characteristics, and to probe possible competitors', present and future, systems weaknesses and strengths. The outcomes of this analysis are the operational objectives which the new system must achieve in order for the further development to be valid. These objectives should not be confused as requirements, as they will change in the upcoming

multiple iterations when deciding the balance between performance, cost and quality. Operational objectives address the end state of the system and what it will accomplish in the large sense. [3]

Functional analysis translates these operational objectives into initial system functions. This analysis focuses on functional characteristics that are needed for the system to perform better than its predecessor or the advances gained with new functionality of implementing advantageous technology, depending on the initial need. To achieve success in defining initial system functions it is often recommended to tentatively visualize the entire system life cycle. Also, if a predecessor system is presented, it is recommended to look for functional characteristics that could be implemented in the new system from the predecessor, since they help validating the case for further development later on. After visualization of top-level functions, it is needed to allocate them into subfunctions to demonstrate a valid way to achieve the desired objectives with a combination of subsystem functions. [3]

Feasibility analysis visualizes the subsystems and defines an early feasible concept. Because the main objective of the needs analysis phase is to validate the operational objectives, no design decisions should be made in defining an early feasible concept, only the idea of how the allocated subsystem functions are implemented. The issues with choosing an optimal design will be handled later on in the development phase. If a predecessor system is present, it might be possible to incorporate parts of its subsystem in a modified form to the new system. In some cases, this might be strongly preferred because of existing support structure for an operational function, such as technical support. This is useful when building a case for the system's feasibility because it helps to determine the costs of the new system, versus the predecessor system, more accurately by, for example, in terms of maintenance costs. If the new system uses new technology, of which there are no existing real-world applications to use as reference, the case must be built on theoretical and experimental data available that has been done on the candidate technology. The feasible system concept needs to address the major aspects of the system: development process and strategy, risks, design and evaluation method. [3]

Needs validation is the final phase which validates the results of previous steps. The goal is to formulate the case for the new system that meets the initial need with an affordable cost and at an acceptable risk. This is achieved by designing a measure of effectiveness and an effectiveness model. Measure of effectiveness (MOE) identifies the characteristics that are critical to the operation of the system and defines metrics which are important when establishing requirements and testing the system. Effectiveness model contains a set of scenarios which are possible for the system to encounter in its life span. These two are combined in needs validation to determine whether a system concept is feasible and satisfies the operational objectives set by the projected need. The operational objectives are, in the validation phase, transformed into operational re-

quirements which, as stated before, define what scenarios the designed system must be able to perform. From here on, these operational requirements are a reference, to which all subsequent initiated development will be judged against, and because of that should be clear and consistent. [3]

2.1.2 Concept exploration

The requirements defined in the needs analysis are often broad and vague, since they are used to justify the initiation of new system development. Concept exploration takes operational requirements from needs analysis as input and refines them into more engineering-oriented requirements, in this thesis referred as system performance requirements. Operational requirements answer the question “why should a new system be developed” when system performance requirements address “what should the system do” and “with what performance”. This conversion is needed for deciding a viable system concept, functional- and physical-wise, in the concept definition phase later on. This phase will also present a set of potential system concepts which fulfill the system performance requirements. Concept exploration can be divided into four main actions: Requirements analysis, performance requirements formulation, implementation concept exploration and performance requirements validation. The complete flow diagram of concept exploration by Kossiakoff et. al. [3, p. 170] is modified such that represents the structure of this thesis and presented in Figure 7.

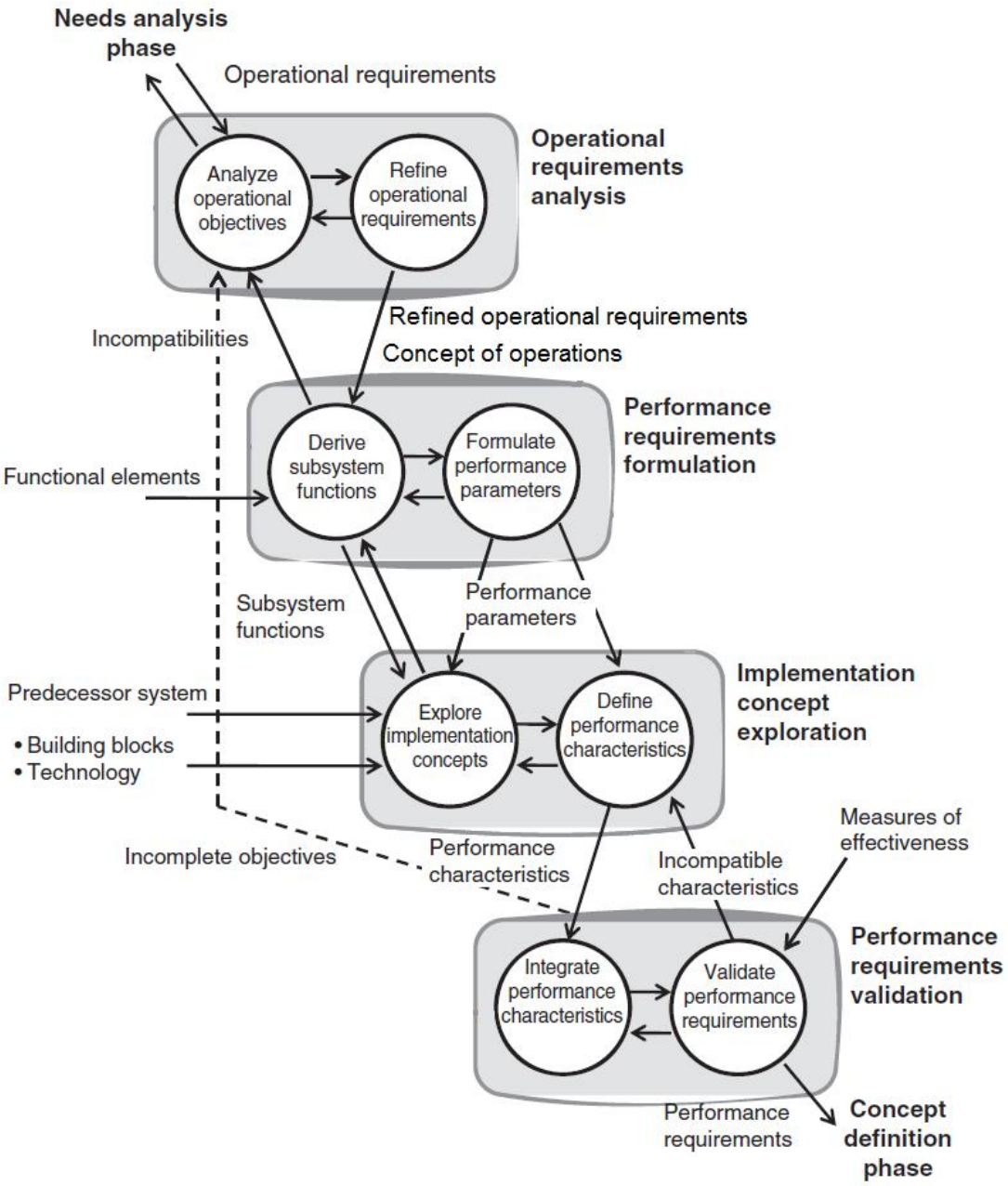

Figure 7. Concept exploration flow diagram.

Requirements analysis refines the requirements set by needs analysis into more engineering and system-oriented requirements. The operational requirements from needs analysis are usually expressed in the context of a fixed user or operator, and because of that might contain requirements that are redundant with other requirements or technologically infeasible. Kossiakoff et al. [3, p. 172] suggests that for each requirement, a set of test questions are applied. These questions, which are presented below, are a baseline which determines if a requirement is valid.

1. Is the requirement traceable to user need or operational requirement?
2. Is the requirement redundant with any other requirement?

3. Is the requirement consistent with other requirements?
4. Is the requirement unambiguous?
5. Is the requirement technologically feasible?
6. Is the requirement affordable?
7. Is the requirement verifiable?

If any requirement gives the answer “no” to the questions above, it needs to be refined and revised against other requirements. After the individual requirements test, the whole set of requirements needs to be verified as one and checked that they cover all the user needs and operational requirements, and also are feasible in terms of planned schedule and cost. In addition to the refined operational requirements, a concept of operations (CONOPS) should also be defined [3]. CONOPS defines a general system approach and should be considered as an extension to operational requirements. CONOPS is usually prepared by the customer and it helps determining how the system should be developed and who should be the actor, which clarifies the main goal of the system [3]. Because CONOPS is considered as an extension to operational requirements, it shouldn’t bind the concept into a specific implementation. In this thesis, CONOPS is presented as use cases. Use cases are used to model the system’s functional requirements using sequence diagrams to describe the sequence of interactions between participating actors [11, p. 11].

Performance requirements formulation continues the functional analysis, started in the needs analysis phase, by transforming the refined operational requirements into more in-depth subsystem functions. The goal is to explore alternative functional concepts by visualizing subcomponents and identifying requirements for subsystems. In this phase, it is recommended to recognize the system’s inputs and outputs to furthermore understand and categorize subsystems via transformative functions [3, pp. 179-182]. Transformative functions use signal, data, material and energy inputs to produce the projected end state of outputs set by the operational requirements. From each subsystem function, performance parameters are formed which measure the performance of each function individually. These performance parameters are aggregated later on into performance characteristics, and finally into measuring the performance of the system as whole.

Exploration of implementation concept phase can also be perceived as a physical definition phase because the main emphasis in this phase is on exploring feasible technological and physical system concepts, and refining performance characteristics from performance parameters. The functional concepts explored in the previous phase can have a very different physical implementation in real-world-applications and use multiple technological concepts that vary greatly from each other. If parts of a predecessor system or its subsystems are to be used, the advantageous technologies and concepts that improve these specific parts should be explored first. This is because it is often easier to assess the advantages in performance, development risks and cost on an already existing

system or subsystem, as mentioned above. However, it should be noted that using parts from a predecessor system can severely limit the systems growth potential. This phenomenon relates to the natural temptation of an engineer to take the easiest way possible, which usually means modernizing a part of a predecessor system, and disregarding the extra work in investigating advantageous new technology. Innovative concepts that use advanced technology generally do pose a greater risk in terms of development and cost, which originates from lack of empirical data and experience gained from using these advantageous methods.

The main goal for performance characteristics is to refine and identify those performance parameters that are imperative and essential to the primary mission of the system. The number of performance parameters in a complex system can be large, depending on the number of allocated functions in the previous phase, and often include many that are only useful for testing or developing the system. This especially occurs when the parts used from a predecessor system are derived from a totally different application. The performance characteristics of interfaces, and everything with user interaction, are especially important as they often place physical constraints to the concept. [3]

Performance requirements validation combines the performance characteristics from alternate concepts refined in the previous phase with effectiveness analysis, which validates that the combined performance characteristics define a system that will possess the projected operational requirements, and produces a set of system performance characteristics. Even though this evaluation of characteristics has been ideally done in the previous phases, there might still be redundant characteristics, or the introduction of an important evaluation criterion in the more detailed effectiveness analysis, that highlights deficiencies in the explored feasible system concepts. The system performance characteristics address “what the system must be able to do and with what performance” in engineering terms that can be verified analytically or empirically. These characteristics are then transformed into a requirements document, which is called system performance requirements in this thesis. The requirements document needs to be done thoroughly and with care because presented alternative system concepts from this exploration phase are evaluated against this document. The document depicts the system architecture down to the subsystem level but should still be considered as a living document, which will be updated during the development process. [3]

To create a proper system requirements document, that is understandable and accurate, is a challenging task. Usually the parties taking part into creating the requirements documents don’t fully understand each other [12]. The requirements listed in the document should be distinguished from being essential or desirable features. Desirable features provide extra performance and might provide greater fault tolerance in abnormal circumstances. Essential features are needed to accomplish the main goal of the system set by the projected need. Most of the requirements listed by the customer are often depict-

ed as essential, even though they might be desirable features. Also, the requirements shouldn’t define how the system should be constructed, as that will be defined later on.

2.1.3 Concept definition

The concept definition phase is where the actual development of a system starts. The preceding phases have been concerned with designing a system to a level necessary to define feasibility and to build a case for further development of a system. In this phase, it is necessary to define a system so that its performance, development effort and cost can be estimated and compared with the predecessor, or similar, systems. For the preferred system concept, a development schedule and system specifications are also defined. The phase consists of three main actions: Performance requirements analysis, functional analysis merged with concept selection and concept validation. The complete flow diagram of concept definition by Kossiakoff et al. [3, p. 202] is modified such that represents the structure of this thesis and presented in Figure 8.

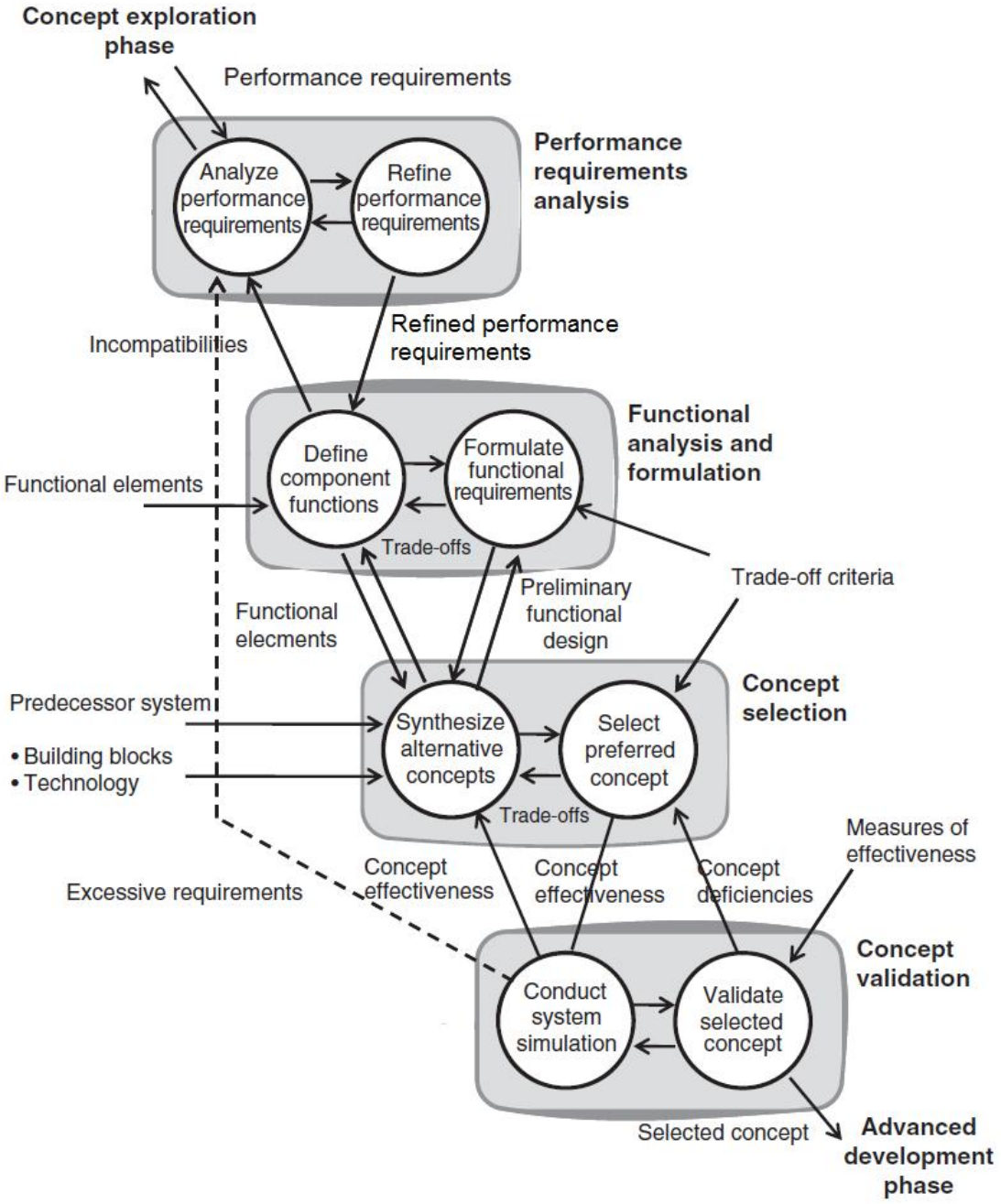

Figure 8. Concept definition flow diagram.

Performance requirements analysis ensures that the planned operating site and other auxiliary activities are in consensus with each other. As with the preceding phases, it's first necessary to analyze and refine the output from previous phase, removing possible redundant requirements. Even though the requirements have been ideally perfected in the previous phases, the requirements are often influenced by system engineer's personal presumptions of what goals are easy and what difficult to attain. Functional and performance types of requirements are usually well defined, since they interact and affect the system's main goal directly. The requirements usually omitted are related to compatibility, reliability and environmental requirements. Compatibility and environmental

requirements relate to the operational system, how the interfaces co-act within the operating site and what kind of environmental phenomena, natural or man-made, the system must be able to withstand. Reliability is often omitted because it's defined more accurately in the deployment phase in co-ordination with the customer; how the system will be serviced and what the length of the guarantee period is. If there is a set date from the customer at which the system must be ready for operation, it's important to understand the priority of meeting it in the given time limit for development, and relative to system development cost. Also, all the requirements after this phase should be verifiable, which means that in order for a requirement to be useful and for the systems engineer to validate that the requirement has been fulfilled, it needs to be measurable. Unquantifiable requirements usually relate to UI design, with requirements such as "user friendly" and "modern design". This is due to the fact that users with different backgrounds have very distinct views of interfaces and usability.

In functional analysis and formulation, the refined performance requirements are used together with functions allocated in the concept exploration phase to define them into components that can be used for comprehensive comparison between alternative concepts. In software-intensive systems, the functional allocation in terms of components can be difficult due to the level of abstraction and absence of common functional elements that can be easily perceived, such as motors and tanks. This phase is very closely related to the concept selection phase and it's difficult to draw a line between them, as it is needed to create preliminary functional design and define functionality for components from the alternative system concepts explored. The functional design is needed to formulate trade-offs between the functional requirements and their functional components, as well as the trade-off between the synthesized alternative concepts.

A trade-off depicts a situation where losing one aspect gains another aspect in return. The trade-off analysis should consider the most important system performance requirements in terms of system cost, performance, deployment schedule and projected need, and their individual and combined risks. As stated previously in the concept exploration phase, all explored possible alternatives, and not only the most obvious and appealing concepts, should be reviewed to demonstrate a thorough analysis. To establish a valid comparison between alternate concepts and their aspects, weighing factors for each selection criteria must be defined. The purpose of weighing factors is to highlight the critical criterions which affect the total value of the concept and to identify the less critical, optional aspects. When weights for system objectives have been defined, it is important to perform a sensitivity analysis on the weighted variables. Sensitivity analysis determines how robust a system is in terms of variable variation. For example, if the importance of production rate of a system becomes 30% more important than system stability due to fluctuations in global economy, or some aspect of the system becomes obsolete due to new technological advances. The objective of trade-off analysis is to or-

ganize the alternative system concepts in order of superiority in terms of combined development and deployment gain and risk.

Concept selection, after the trade-off analysis is complete, is often complex because the analysis produces multiple system concepts with similar total value and involves the comparison of aspects that cannot be defined by an absolute value. Before selection, the number of concept alternatives should be small enough to make a valid decision, as large number of alternatives with equal value usually indicates a coarse trade-off analysis between concepts and thus should be revised. Each candidate should be evaluated against the list of system performance requirements, possibly adjusting concepts which reject some critical criterion, and result estimated if it is in balance in terms of cost, risk and schedule. The goal for concept selection is to choose a preferred system concept, which will be validated by system simulation in the next phase. The chosen concept can, and should if major deficiencies rise, be evaluated again after the simulations, but it is practical to vary between a few system concepts at this point because the schedule is often tight in modern system delivery projects.

Concept validation is the final phase before initiating system architectural development, which validates the chosen preferred concept, weighing and validating the concept deficiencies and effectiveness by conducting a system simulation. The system concept simulation provides the system engineer with information of possible deficiencies in system characteristics that need revising or excessive system requirements that cannot be implemented in the preferred system concept. Simulation also establishes how robust the deployed system would be by varying parameters to simulate environmental and system variations. For complex systems, simulations can nowadays be done with relative ease with the help of computers and advanced simulation applications that include multiple built-in models which make predicting system behavior accurate. If thorough simulation for the system behavior cannot be conducted, at least a critical experiment which depicts the main functionalities of the preferred design should be conducted to make a solid case for future development. Analytic analysis at this point of the development process is not often enough to validate the selection of preferred concept, especially if there is no predecessor system, or subsystems implemented from it, present to establish solid value for development risk.

The main goal for the concept definition phase is to define a system concept that fulfills the system performance requirements such that the future development is justified. To validate the chosen system concept, a project plan describing how the engineering phase is managed is essential. The project plan consists of system development schedule, which states clearly, to all participants, what are the timescales and objectives for each task in the development plan. The development plan describes all the functions the system must be designed to fulfill the system performance requirements and address the

risk in achieving them. A set of system functional specifications are usually derived to help describing system functions and architecture.

2.1.4 Advanced development

The advanced development stage can be perceived as an early development stage for the already chosen system concept. Often the chosen test concept at this stage has only been validated by simulations and no actual development for the physical aspect of the concept, hardware- or software-wise, has been done. The main goal for this phase is to define the major uncertainties and risks identified in the previous analysis phases, to determine their actual impact on a real system and resolve them through analysis and development work. This stage is important for the to-be-deployed system, as the ignored unresolved risks in this stage that realize in the later development stages may have an amplified effect, for example in hardware decisions, and severely degrade deployment schedule. It should be noted that not all systems should go through the advanced development stage. This is the case especially in a system development where a predecessor system is present, and numerous subsystems and functionalities are implemented from it with well known, and reliably predictable, characteristics. As with the previous phases, advanced development can also be divided into four main actions: Requirements analysis, functional analysis and design, prototype development and development testing. The complete flow diagram of advanced development by Kossiakoff et al. [3, p. 321] is modified such that represents the structure of this thesis and presented in Figure 9.

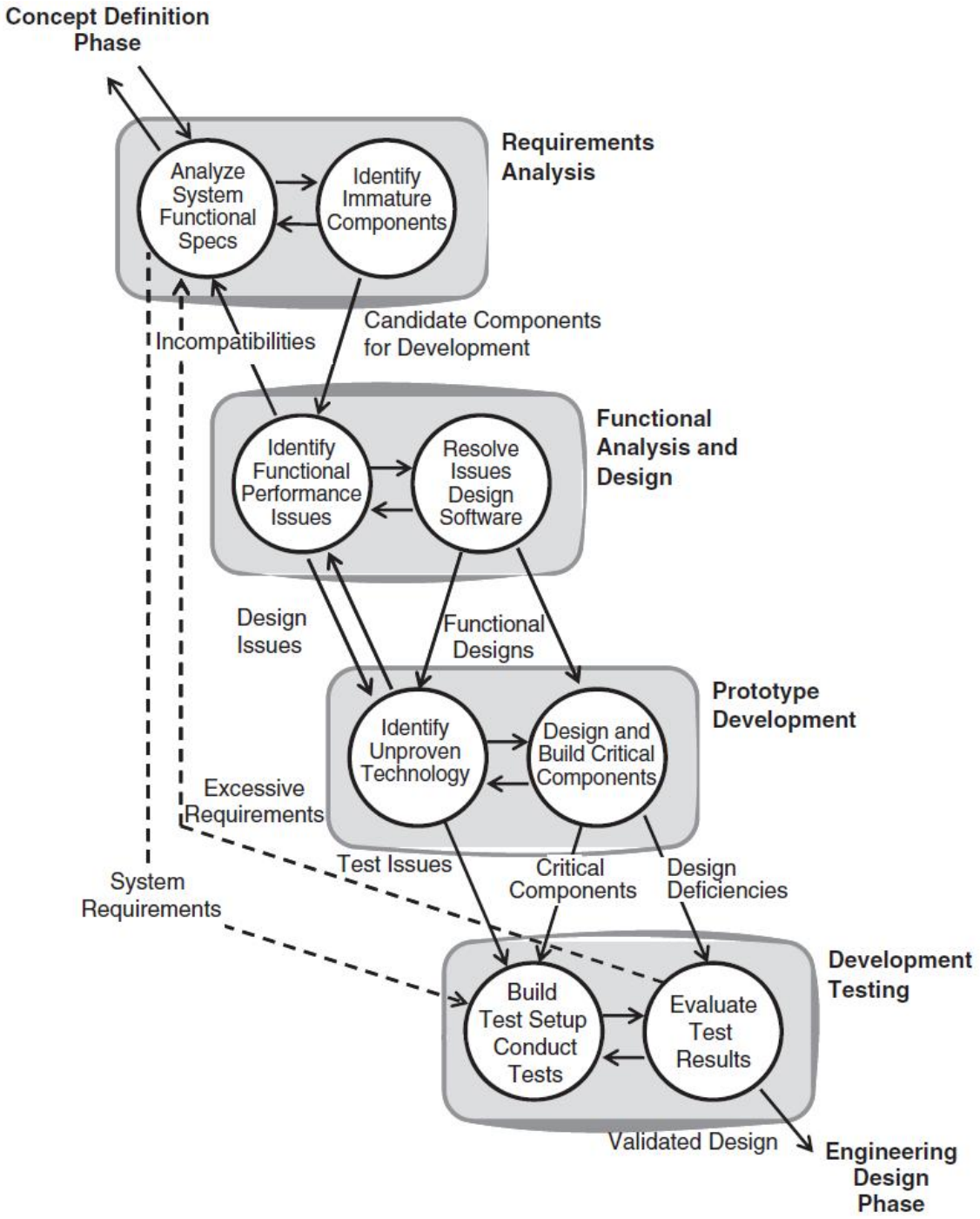

Figure 9. Flow diagram of advanced development phase.

Requirement analysis reviews the selected concept in terms of composed system requirement specifications. The aim in the analysis is to identify system specifications that are sensitive to design parameter variations during the system's life cycle. As said previously, UI requirements tend to be immeasurable and thus these need to be recognized as possible risks in choosing system building blocks. The wrong UI technology can lead to interface and upkeep problems in the implementation phase that aren't, or for some reason cannot be, taken into account. The system requirements that aren't readily solved by the chosen system concept should be the main issue in the requirement analysis, as

these requirements usually include components that haven't been, or are loosely, defined and thus inherit a great development risk. If there is a predecessor system present, from which new system requirements have been derived, it's important to understand the new implemented functionality, its difference to the deficient predecessor, and the possible issues that might rise from utilizing it. There are usually three types of components in systems that are in need for development: Ones that perform complex functions, ones that have performance requirements over proven limits and ones that have interactions with the environments that aren't fully understood. The end product of this analysis should be a set of identified immature components that need revising and development.

The components identified in the previous action contain, or lack, some functionality that contradicts or doesn't meet the system performance requirements. These issues need to be resolved in the function analysis and design phase. The issues usually rise from the fact that the increased performance and functionality gained from the new system concept results in increased complexity in the deployed automation system [3, p. 327]. In the functional analysis phase and design phase, it is important to examine the subsystems, their components, and interactions between them. This is to validate that each component can be designed, built, tested and assembled independently and, more importantly, integrated with other components with minimal adjustment and fitting. New software components are often the ones that need testing and validation in addition to an analytic analysis, since they are far too complex and usually control low-level hardware components, such as valves and motors. The outcome should be functional designs that are ready to be built and validated in the next, prototype development phase.

The prototype development phase is a crucial stage for new system concepts that have no predecessor system present, as it is the main risk mitigation technique in the development process. In the previous two phases, requirements and functional analysis, the system engineer has identified components that need development and prepared functional designs that are able to validate and identify possible deficient technologies and excessive performance requirements. It should be taken into notice that the prototype doesn't address issues with physical characteristics in the deployed system, such as fatigue cracking, since these issues usually require the whole system to be built in for them to manifest. The main goal of this phase is to build the critical components that make the framework for the system concept and include the most important new functionalities that led to choosing this concept. This is to mitigate program risk by building a working practical application. The design deficiencies that rise in the building of critical components should be evaluated and possible reflected back to the functional analysis for revising and possible modification. The output of prototype development should be critical components that need to development and validation.

The final phase of the advanced development is building the test setup, including external simulators and interfaces to induce a controller environment, and conducting tests specified in the master test plan. To determine that the design issues have been resolved, a systematic analysis in form of a documented test program is needed. The purpose of the test program is to determine if the system design is mature enough to proceed to the engineering phase. The starting point for creating the test program should be determining the objectives; testing the subsystems against performance requirements, uncovering design flaws in high risk systems, validating the system performance publicly, validating interfaces with chosen inputs and possible increasing customer's confidence in a particular aspect of the system. After determining the objectives, a review of the operational and performance requirements is needed to define the key features and parameters that need to be included the test plan, as usually it is not possible to test every requirement listed due to project schedule. With the key parameters defined, it is needed to define which inputs stimulate the desired components and which outputs measure the response. The measurements often need external test equipment, so that they can be reliably determined, which need also reviewing in case of possible interface modifications. Now the test cases, which test the key aspects of the system, can be generated. Finally a test schedule should be developed, which addresses the preparation, testing and analysis of the test cases and also the needed manpower and cost of the test program.

The evaluation of test results can be done in real time with predicted values from a simulated model, or the values can be recorded for a more in-depth subsequent analysis. If a defect is found during testing, it should always be double-checked that it isn't caused by a defect in the test equipment or by an illegal procedure during testing. This is because the test equipment is usually composed in less time than it took to design the system component, which makes it prone to deficiencies. The evaluation of UI's is particularly difficult since the success of the test case depends greatly on the interaction of the operator and the system. The main problem in the evaluation of UI's derives from the incapability of objective quantitative measurement, as stated before. However, the UI should still be evaluated at least in four aspects: ease of usability in terms of operational controls, clarity of visual graphs and displays, amount of useful and useless information presented and the amount of user assistance in case of fault situations and alarms.

The output of the advanced development should be the validated design which has all the major uncertainties resolved. The most difficult part in the evaluation is to decide when the components are developed and tested to a level where they can be transferred to the engineering phase. The main goal of advanced development phase was to mitigate risks in form of a prototype, and because of that exhaustive system design is not practical. Usually, in the actual projects, the priority of the system development is to start the full-scale engineering phase with a maximal expectation of success as soon as possible.

With this said the system engineer's task is to decide when the system design is mature enough to proceed into engineering design phase.

2.1.5 Engineering design

The engineering design phase is concerned with integrating all the component parts of the system into a fully operating concept which can perform and fulfill the set operational requirements. The previous phases have already established the conceptual designs and components that are required to achieve the requirements, but the engineering phase begins to specify the needed internal and external interfaces for the system and also implements the first hardware and software components. The engineering phase is closely bound to the integration and evaluation phase since it provides, in addition to fully engineered components, the detailed test and evaluations plans which are used in an iterative manner to improve the engineered design. As with all the preceding phases, engineering design consists of four main actions: Requirements analysis, functional analysis and design, component design and design validation. These actions are depicted in Figure 10.

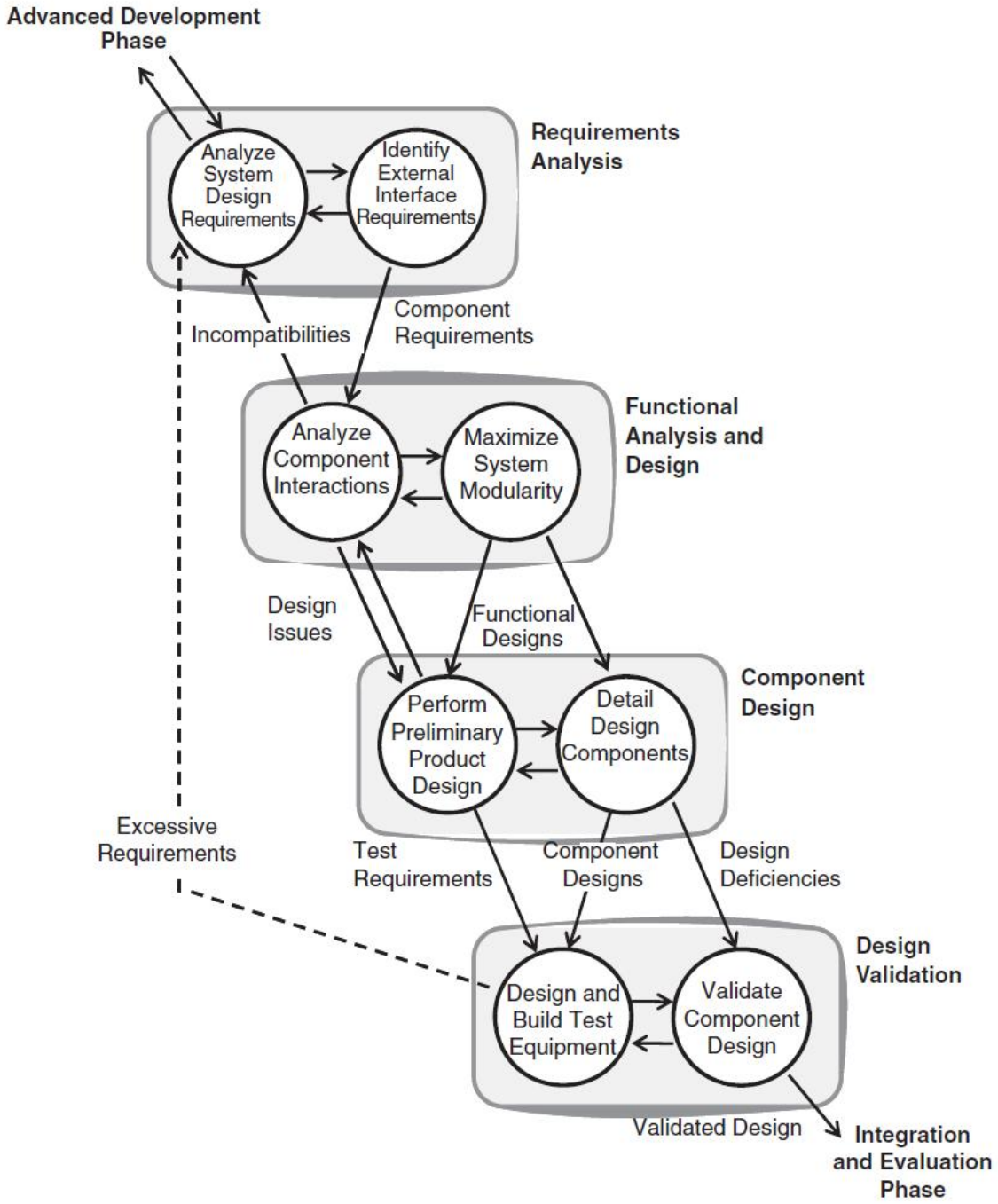

Figure 10. Flow diagram of engineering design phase. [3, p. 413]

The requirements analysis further analyzes the system design requirements specified in the prototype development for their consistency. However, the already resolved design issues in the prototype should be carefully reviewed to ensure no residual risks remain. The main emphasis in the analysis is on identifying the needed interactions for internal and external interfaces. The requirements of external interfaces should be considered carefully since the entire system hasn't been assembled before this stage and the needs of different system components might contradict each other. One of these crucial external interfaces is the UI operated by the intended user. Prototypes of the UI, user consoles and displays, should ideally been created and evaluated in the advanced development stage. If not, the preliminary evaluation should take place as early in the engineer-

ing phase as possible. The importance of internal interactions between modules rises in this phase because the modules are integrated into an operating system for the first time and might highlight risks that haven't been identified before. This is due to the fact that it is rarely possible to perform, even preliminary, integration tests in the advanced development stage. The output of this stage should be the identified requirements of the internal and external interfaces, such that the functional designs can be derived from them.

The functional analysis and design phase takes the requirements of the interfaces as input and refines them into more simple interactions of system components and the environment. The purpose of this phase is to arrange the functional components in a manner that maximizes the system's modularity. This will ease the development process in the future, especially the testing phase, when the dependencies between subsystems is minimized allowing each component to be developed, design, built and tested as a separate unit. It will also clarify the internal interface issues because the interconnections between modules have been simplified. However, simplifying interconnections shouldn't lose any information; the goal is to only generalize and group entities. In software-intensive systems, the amount of modularity achieved in component design is closely bound to the level of abstraction used.

To achieve these goals, the functional elements should be significant, singular and common. Significant in this case means that the element performs a significant functionality in the system, which is distinct from other elements. Singular element means that the functionality is technically within one engineering discipline, for example safety engineering. A common element means that the functionality performed can be found from multiple, already deployed and operational, systems. The most obvious functional element that fulfills all three aspects, which is a part of almost all the systems, is the UI. The output of this phase should be the functionalities arranged in a modular fashion, ready to initiate the component designing phase.

Component design phase is the main phase in charge of assembling and shaping the detailed functionalities that haven't been engineered yet. This phase consists of iterative procedure between preliminary and detailed design that allows the implementation of modular functionalities into hardware and software packages coupled with interfaces to prepare the module for upcoming unit and integration testing.

Preliminary design provides the framework for detailed design, including specifications of interface and design, and also the plans for further integration and testing. This includes all the issues not resolved in the previous development stages, which is usually due to the issue being perceived as insignificant at the time or being discovered after the prototype simulations.

Detailed design matures the components up to a level where they fulfill the task and perform functionality to which they were planned for. The typical products from this phase are drafts, engineering and interface control drawing specifications for production and detailed test plans coupled with quality assurance plan. Test plans and quality assurance plans should be considered in the sense of reliability in this phase, to ensure the engineered components will perform the intended functions correctly for the planned life span of the system under specified environmental conditions. [3]

In software intensive systems, reliability differs greatly from hardware reliability. Hardware failures are usually due to components wearing out or short-circuiting which is due to extended usage or stressed environment. Software doesn't wear out or suddenly stop, instead these faults are often caused by the hardware on which the software is built on. However, software does cause some failures and crashes due to imperfections in the coding. These failures are usually caused by unexpected variable states manifesting in the program which then produces erroneous output with seemingly steady input. In software, a defect is defined as a binary variable meaning the program either works as intended or it doesn't. In the software industry the six sigma defect level is 3.4 defects per million lines of code [13]. Often when the software contains multiple functional defects, it is due to errors in understanding the requirements placed by the customer. If the software development process doesn't address requirements quality, it is bound to produce poor-quality software [13, p. 6].

Validation phase consists of fabricating the needed design components up to a level where testing equipment can be designed. To validate the components designed and constructed, it's essential to perform unit qualification tests for each modular component. The components, up to this point, should have been at least preliminarily tested during the development process to avoid issues rising during unit testing. These issues often relate to interface conflicts, which severely interfere with the testing procedure. If a prototype has been built in the advanced development stage, its test methodology should be analyzed if it can be used directly, or in modified form, to test the components in this phase. It should be noted that this phase is considered as development testing, not acceptance testing, since the design flaws will be iteratively resolved, whereas acceptance testing either accepts or rejects a component. The validation testing done in this phase should concentrate only on the modular entities, not their possible interconnections. [3]

2.1.6 Integration and evaluation

When the modular system blocks have been engineered and validated separately, it's time to qualify the design for operational use. Ideally, integration and evaluation for the engineered components should be straightforward as the modules and their design has been validated in the previous phases. In real-world applications, all the aspects of the

system cannot be addressed during development and multiple issues rise during system integration. This is to be expected in large and small-scale projects, and the risks can be mitigated with early test planning and preparation.

The success of this phase depends on multiple factors. Majority of the outcome depends on advanced planning in the engineering phase. This is because the integration phase is a separate action from development testing, as the used approach for validation testing depends greatly on how the component is implemented. The planning usually has to be done visually and without anything concrete to reflect on. This is due to the simulation of the system's operational environment often being expensive and time consuming. [3]

The preparation of test plans should start from reviewing the system's operational and functional requirements. This is to ensure no changes have occurred in the engineering design phase which might have an impact in designing the test setup. The changes often are due to alterations in customer requirements, in utilized technology or in program plans. Changes in program plans can be caused by for example, inadequate funding due to fluctuations in the components prices.

The integration phase can be perceived as synthesizing the system into a whole and functional entity, which includes actions to solve remaining issues in component interfaces and interactions. According to Kossiakoff et al. [3, pp. 476-478], a typical test configuration in the integration phase consists of:

- System element, component or subsystem, under test
- Physical element of the component or subsystem
- An input generator to provide the test stimuli
- Output analyzer to measure the generated test responses
- System control and performance analyzers

The failures that arise during the testing are not usually due to the component under test being deficient, but rather the test equipment being inadequate. Also, test procedures executed with inadequately defined test plans and procedures operated by personnel without sufficient training often produce false positive results during testing. These said issues should be mitigated to a level as low as possible, since false positives are hard to distinguish from real errors during the testing and often arise as system instability after the system has been deployed and is in operation.

To evaluate the test results, a system performance model must be generated so that valid predictions from the tests can be established. This model can be ported from the previous phases, with much emphasis on the results from the prototype created in the advanced development stage. If no prototype was created, the preliminary simulations in

concept definition together with system performance requirements should provide enough information to create a valid predictive performance model.

2.2 Interview method

Data can be collected in several different ways, and one appropriate method is by interviews. The reason for having interviews is to “obtain valid information from the most appropriate person” and they demand real interaction between the interviewee and interviewer to be beneficial [14, p. 134]. To perform interviews efficiently the interviewer needs to research the interviewee’s background, values, expectations and skills beforehand. According to Ghauri and Grønhaug [14], there are three types of interviews: Structured or survey research, unstructured and semi-structured.

Structured interview is an interview where the same questions and standard format is used for all the interviewees’ and the emphasis is on systematic sampling and statistical methods. The advantages of structured interviews are in repeatability, other researchers can replicate the interviews under similar situations, and obtaining wide range of applicant information [15]. However, a highly structured interview can severely restrain the information flow by requiring interviewers ask the exactly same questions in predefined order [16].

In unstructured interview the questions are rarely specified in advance and the interviewee has full liberty to discuss and give opinions on the particular question. The interviewer’s task is to lead questions and record responses for later analysis. This is advantageous in the sense of discovering entirely new information. The ability to probe is much greater than in structured interviews and also allows the interviewer to return to a specific topic and elaborate upon it [17, p. 146]. The disadvantage is that the lack of standardization might cause the interviews to be less reliable than structured interviews due to information becoming tainted in uncontrolled interview conditions [15, p. 602].

Semi-structured interviews differ from unstructured such that the topics, issues, sample size and questions to be asked, for the most part, have been determined beforehand. This is useful when the interviewer has a specific topic for their interviews in addition to a range of other methods embedded in the research [17, p. 135]. Because the topic in this thesis is about software development and automated testing but the amount of information from the interviewees is unknown, the emphasis is on semi-structured interviews.

When preparing for an interview, Ghauri and Grønhaug [14, pp. 133-134] suggest three steps are taken: Analyze the research problems, realize what information is needed from

an interviewee and search for people who might be able to provide that information. After these steps, drafted interview questions should be compared to the research questions constantly to check whether the interview questions can yield reasonable and valid answers [14].

2.3 Graph theory

This chapter presents the methodology of graph theory and digraphs. It's based on the course material of "MAT-62756 Graph Theory" lectured in Tampere University of Technology [18] and on the book Bang-Jensen, Gutin (2008) *Digraphs: Theory, Algorithms and Applications* [19].

Formally a graph G consists of two finite sets: a set $V(G)$ of elements called vertices and a finite set $A(G)$ of paired distinct vertices called arcs. Consequently a graph $G = (V, A)$, where $V = \{v_1, \dots, v_n\}$ and $A = \{a_1, \dots, a_m\}$, has m arcs and n vertices. If an arc called a_k has two end-vertices, v_i and v_j, it can be expressed as a pair $a_k = (v_i, v_j)$. In an undirected graph, the arcs (v_i, v_j) and (v_j, v_i) are the same. An illustrative example of an undirected graph is presented in Figure 11.

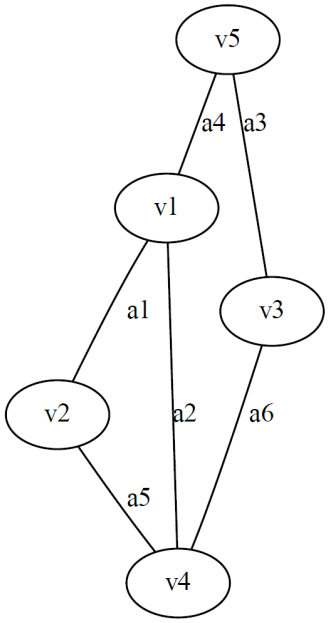

Figure 11. Undirected graph with five vertices and six arcs.

A directed graph, from here on referred as a digraph, is a graph where the arcs connecting the vertices have a direction. Thus the arc set $A(G)$ consists of ordered pairs, where an arc $a_k = (v_i, v_j)$ means the arc leaves v_i and enters v_j. In an arc (v_i, v_j) the first vertex v_i is called as its tail and the second vertex v_j as its head. It can also be said that

v_j is dominated by v_i. An example of a digraph is presented in figure 9, which uses the same set of vertices and arcs as Figure 12.

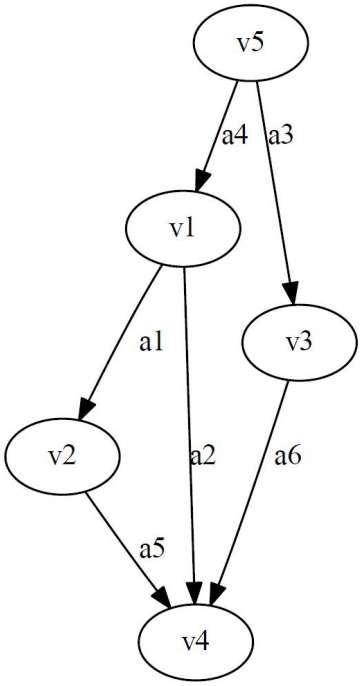

Figure 12. Digraph with five vertices and six arcs.

3. PROOF OF CONCEPT

The following chapter describes how the V-model from systems engineering viewpoint was used to develop test automation for an automation system application. The automation system platform used in this thesis is MetsoDNA, developed by Metso Automation Oy [20]. The system under test (SUT) is a paper machine's machine direction (MD) control system. It should be noted that the SUT has already went through V-model's decomposition and definition phase and the developed test automation will fill out the integration and validation phase. This means that the system validation phase for the test automation is actually the integration and evaluation phase for the MD control system.

3.1 The need

In automation system application research and development (R&D), the amount of automated testing has been on the rise for the past years due to increase in the number of version management and continuous integration (CI) platforms developed, which can be used without major modifications for a wide range of automation applications and platforms. But for a specific application group, there rarely are tools for integration testing that can test the product groups thoroughly with reasonable accuracy and with a good return over investment (ROI) in terms of work hours. This is due to the amount of work hours involved to develop test automation in relation to gained benefit. That is why automation system manufacturers don't have interests to develop extensive test automation; the goal is have an easy to use, comprehensible and open source based test application in a compact package that is easy to maintain [21]. The deployed automation systems are increasingly becoming more customer-focused making the task of developing comprehensive test automation, for a wide range of automation applications that often have a long lifespan, difficult. Unit testing, which involves the validation of functional design in a single system component or module, is usually well maintained since it's done by the coders or developers themselves when developing a new feature or product [22].

The main problem in developing test automation for automation systems is that the architecture and interfaces used can differ greatly between manufacturers. This is due to the fact that there hasn't been a solid standard for building an automation system technology-wise. Early automation systems didn't use an international standard when the function block diagram (FBD) programming language was implemented; it was the industry's standard way of implementing commands back then. In the past there have been some parts in automation systems that include portions, for example a develop-

ment system, under an international standard. An example of this is the CODESYS development system that utilizes the standard 61131 from International Electrotechnical Commission (IEC). This development system has been used in some automation systems. The international standard IEC 61131 applies for programmable logic controllers (PLC) and their peripherals that have their designated use in controlling and commanding industrial processes [23]. The standard's third part, IEC 61131-3, defines basics for programming languages in PLC's, for example syntaxes, operating systems and testing methods.

Other than the said IEC standard, the automation system manufactures have had a free hand in system architecture and deployment technology-wise. For test automation this means that for it to maximize the ROI in terms of work hours the system should be developed by, or in close co-ordination with, the original developer of the automation system who has in-depth information about the structure and technologies used. In this thesis, the test automation was developed for the R&D department of Metso Automation Oy to support the development work of new products and features for the MD control system. [5]

The main task for the test automation system is to assist knowledge workers in developing new features or products by validating the integrity of the current development version and assuring the quality of software before release. Ideally, the test automation system should be dynamic in the sense that it can be used to test single features with a few modules as well as entire product groups with multiple features in correlation with each other. There are no predecessor systems currently present for this application scope so the development of test automation is needs driven. The knowledge workers referenced in this thesis are engineers in the R&D department who have at least 10 years of experience in the field. [5]

To develop a new feature or product, a sandbox environment which has the latest official versions of all the applications is needed initially. Developers create unit tests in parallel with the development of a new feature, which will be added to the master test case list containing all the test cases. When the new feature is functional, it will be first tested with unit tests created during the development by the developer. After this, the developer will test the new feature with other products against a simulated customer case. All these tests should be able to run independently from the user, requiring no interaction whatsoever to be completed. When the testing is completed, a report from each test phase is presented to the user. If the testing failed prematurely in the early phases, the user should be let known immediately about the failure. From this life cycle we can identify that the system has three top-level functions: User-interface, test automation core program and master test case list. Master test case list can be divided into subsystems of unit-, sub product- and product testing. Test automation core program can be

divided into interface subsystem, which handles the communication to the automation system, and test application that executes the tests. [5]

For the user to have ease of access, user-interface should be web-server based. This mainly gets rid of the user-side of incompatibility problems since the web page can be operated from any computer with a web browser that has sufficient user access rights. However, a web page can have a totally different look depending on the browser used and even some compatibility problems, though they are largely preventable before-hand [24].

The master test list should be easily updatable by the developers, since they will be adding test cases as a new feature is being developed. Easy updating means that single test cases are easily modified by the user, for example in an Excel-sheet which displays the steps included in the test case, and that they are sorted by their purpose; functional testing, unit testing and qualitative tests. This makes the upkeep of the test cases and their configuration easier.

For the test automation core and its interface subsystem, a widely used and practical programming language should be used so that possible add-ons or modifications are easier to do by someone who hasn't been involved in developing the test automation. This will also help the updating of the core and interface subsystems if the chosen programming language allows new advantageous methods to be implemented in the future. Also, the programming language should be modern in the sense of future support by modern processors. Severe problems can arise if the language becomes obsolete before the projected end of life of the developed test automation system.

The usual workflow in testing a new feature by integration testing consists of: generation of needed modules, checking the integrity of modules to ensure they can be executed, importing them on a process control station (PCS), starting the PCS and initializing test cases. The workflow is depicted in Figure 13. This typical scenario should be done automatically with user only defining what kind of project is used in the tests. The running time of testing one feature shouldn't exceed that of eight hours, so that the tests can be run in one workday.

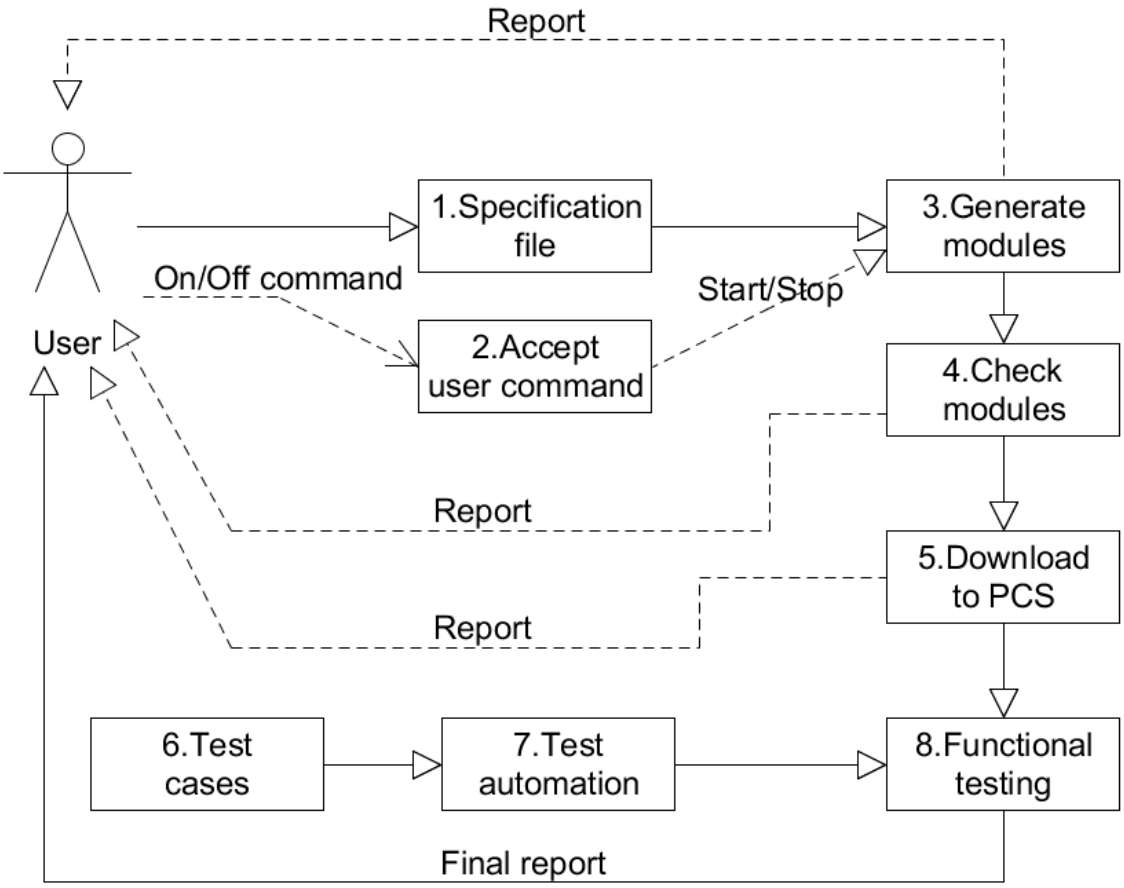

Figure 13. Workflow of testing a new feature.

As mentioned earlier, a thorough test of the entire product group is necessary. The workflow is similar to that of testing one feature except that the generated project is much larger, including many products and features. For this scenario there is no time limit since it can be perceived as very rare, for example before publishing a new version of the product. The execution of a product group test is mainly asynchronous, which in this case means the tests are event based. [5]

All these scenarios should be made easy for the user to initiate. The user should be able to mix scenarios and configure them as he sees best, and also be able to start the test cases with minimal work input. In configuration, the scenarios are linkable so that the user is able to make multiple test case workflows to test a number of features or products. However, if a test in a linked workflow does fail for some reason, the whole test should be stopped and a report sent to the user immediately.

3.2 Concept exploration

The operational requirements listed in the previous chapter were applied to the test questions presented in chapter 2.1.2 to refine them into more engineering-oriented re-

quirements. The operational requirements cover the user-side need without making any commitments to the system architecture.

After refining the operational requirements, it is needed to generate requirements for the subsystems visualized in the previous chapter. As input, the test automation system takes test project specification and transforms it to a functional automation application in a PCS. The first task the system has to do is to generate all the needed products specified in the project specification document. A tool for this already exists, that uses the specifications from the user to generate the needed modules.

When the modules are generated, they are imported into a personal workspace and checked internally and externally by tools which ensure the integrity of the modules, and that all input ports are connected correctly leaving no port undefined. When the modules have been checked, the next step is downloading the module package into an automation system, a PCS which runs the modules, and initiating a startup of the system.

After a successful startup of the PCS, the functional testing of the feature or product can begin, utilizing a communication protocol which allows the test application to read and write positions in the PCS. After all the functional tests, a thorough final report will be given to the user which should also address the previous phases at least briefly, for example time elapsed and possible warnings In summary of the functional definition, the test automation core system has five subsystems: system building tool, workspace, PCS, test application and communication protocol.

The master test list should be in table-form, at least in the user level, to ease the management and modification of test cases. These tablet-form test cases can then later on be transformed to a more suitable form for the test application to read. The table can consist of multiple sub-tables, if the test cases have multiple downstream dependencies.

The web based UI should be compatible with four of the most used web browsers by market share, because with this requirement we cover over 90% of web browsers used globally [25]. The UI should also be user friendly and the layout to be up to date in modern standards so that the configuration of projects is clear and concise for knowledge workers.

From these operational requirements, a numerous use-cases were composed in coordination with the knowledge workers from R&D department. These use-cases depict scenarios, that are divided into operational and functional scenarios, which the planned test automation should be able perform in its life cycle. The use-cases are an extension to the performance requirements, which will be composed later on.

To explore feasible implementation test concepts already used in some other application in the R&D department, a set of interviews were held with knowledge workers who are known to have been researching or developing test automation for automation systems. The interviews were semi-structured, which means the topic and some questions were prepared in advance. This was done by studying documents published to the internal databases by the interviewees about applications and studies related to test automation. The goal of the interviews was to explore what kind of test concepts have been researched and used throughout the R&D department in various applications and projects.

As stated in the needs analysis phase, test automation applications tend to be bound on a specific automation platform due to the lack of standards in the industry, which is why the aim was to research and refine the test concepts gathered from the interviews to achieve a valid starting point in creating the test automation. This is to maximize the degree of success in developing test automation for the R&D department within the time limit of this thesis. Also, another goal was to attain information about possible advantageous technologies that haven't been researched yet, or experimented upon, and could be evaluated in this thesis.

Information gathered from the interviews revealed that a few test automation concepts are in use, which use similar subsystem structure as presented above, but are present in a different software environment. Despite the different software environment, the test concepts have potential to be used, at least as a valid starting point, in the application scope of this thesis. All the user-interfaces used by the applications were web server based solutions, to provide ease of access to users and to simplify user control access rights, and could be operated with common browsers, for example Internet Explorer (IE). Open source based application solutions were used to deploy the web server.

For the test automation core, a few different approaches were used, keyword- and data-driven testing. Data-driven test automation consists of application-specific scripts, coded in the automation tools programming language to modify target systems variable data [26]. Keyword-driven testing typically consists of application-independent automation framework which processes the tests. The tests are processed from a keyword vocabulary that consists of actions the test automation framework is to perform in a test case [26].

All of the test concepts use the same interface protocol to access and command the automation system. As a programming language, Python was widely used in most parts of the systems, with some of the open-source components being C++. The reason for this is Python being very easy, dynamic and intuitive programming language. Python is dynamic in the sense that different types of variables can be inserted into variables without declaring them first, like in C++.

The master test list subsystem was created with two different ways, either with a template based test case list or a comprehensive list of individual test cases. The template based test cases test only modules that are created from a specific template. For this reason, the test cases are only done for modules that have been duplicated from the template numerous times, to maximize ROI-% in terms of work hours. This type of testing is aimed to test that the duplicated modules have been generated and parameterized correctly and are running in the automation system without errors. The individual test case list contains a set of test cases that are independent from each other, which means they can be run in any order and the test result is the same. Each test case is aimed to test a specific functionality or module and can parameterize multiple modules during the test run. [27][21] [22]

Test concepts, which rose as interesting alternatives in the interviews, from outside the R&D department, were also researched [21]. The specification based payload generation for PLCs by McLaughling et al. [28] utilizes black box testing at its most basic level. The algorithm is able to recognize if the probed system is the desired, one for which the malicious payload is designed to, using model-based testing (MBT) approach. Model-based testing is a methodology where the test cases are created from a test model, which is a formal description of desired aspects to be tested [29]. Even though this article is aimed for delivering malicious code, it could be used in test automation which delivers test cases instead of payload.

Initial tryouts for MBT-based testing have been held in previous research projects in the R&D department, but the ROI-% in terms of work hours was poor since it takes great of effort to design a model with enough accuracy that covers a wide scope of products [21]. In practice, it is often more efficient to create multiple smaller and simpler sub models than a single large one [29]. However, it was analyzed that the generation of a model which identifies the SUT, with a sufficient accuracy, is not possible in thesis due to the given time limit.

All these test concepts mentioned focus on testing the functionality of a control system. As for testing the UI, which means that all the pushbuttons, graphs and data boxes function as intended, many commercial options had been researched and tested. The technologies used varied from machine vision to searching for identifiers in the source code, depending on the implementation platform [21][22]. But as of today, no comprehensive UI-testing, for the application scope in thesis, has been developed that has a good ROI-% in terms of work hours. This is due to the fact that most of the test automations for UI's are unstable in terms of change management and thus, need constant maintenance [22].

The level of abstraction in the test cases also highlights problems in UI-testing; the more complex UI-test cases are the more maintenance the test automation needs. The com-

plex UI-test cases have to be done manually which takes time and often results in negative ROI-% in terms of work hours. Due to the risks in change management and defining the level of abstraction the testing of the UI was set as low-priority in this thesis, main emphasis being on the functional testing.

The use-cases and operational requirements were refined using the explored implementation concepts to produce a set of system performance requirements. Some of the use-cases were identified as desirable features, such as extended configuration options for the user regarding test cases and notifications about ongoing test run. From operational requirements, 22 system performance requirements were aggregated. [5]

3.3 Concept definition

The system performance requirements were divided into operational, functional and qualitative requirements. As stated in the previous chapter, the main emphasis is on functional testing of the MD control system, and because of that the requirements needed to be refined and set into an order of importance to reflect the projected need for the R&D department. The operational performance requirements included mainly operator-side requirements that specified the amount of configuration and control the user should have. In the analysis, a few of these requirements were realized as unquantifiable, such as the need for modern design UI and the simplicity of operating the test automation. To cope with these requirements, large importance was set on the possibility of configuring, or providing multiple alternatives, the UI as the user perceived as best.

The reliability requirements listed under the qualitative requirements depend on the implementation platform. Because the system under test in this particular case is purely software, the reliability requirements depend mainly on the hardware platform. The requirements listed stated that the test automation system should be able to operate in a virtual environment, where the amount of disk space, system memory and number of processors is configurable by the user. This follows the trend in software development nowadays, which is to virtualize systems, creating a virtual version of computers, operating systems and even computer networks, under one hardware platform. This may seem as a risk in the case of a hardware failure but many companies that offer virtualization solutions have abundant amount of hardware at their disposal with multiple redundant safety systems implemented to mitigate the risk of test system failure. The virtualization also increases portability since the whole system can be copied to another location on-the-fly if needed.

Should the virtualization not be chosen as the platform for the test automation, a more in-depth analysis for the hardware components is needed since the system is required to run uninterrupted for multiple years without constant maintenance. As for the schedule

requirement, the system concept should be validated, requirements documented and a framework created for the test automation in the time span of this thesis.

As stated in the concept exploration phase, the functionalities of the explored test automation concepts differ mainly in the utilization of test cases. It is important to understand the functional difference of these two explored test concepts, as they are originally aimed for very different tasks. In the keyword-driven method, dividing of test cases in regarding from what template module they were created is effective when the number of duplicated modules is large enough. The exact number of modules depends on the amount of work needed to generate a test case for a template, which should manifest in a positive ROI-% in term of work hours. The modules in charge of safety, for example interlocking modules, are an exception to this. If the effect of failure in an interlocking module poses a tremendous risk for the system's safe operation, a test case for these templates should be considered even if there are only a few present in the system. The template test case method is very useful in system acceptance testing, where the number of duplicated modules is large. The level of abstraction in the test cases is an important factor since the duplicated modules can perform different tasks, depending on the parameterization, and cannot often be, for example, tested with absolute output values. One possible way around this problem is to use relational operators; greater than and less than.

The second test concept that uses a data-driven method, containing a test case library that consists of a number of individual test cases which the user can run in arbitrary order, is aimed for testing a specific functionality or aspect of a feature which is useful in continuous software development and deployment testing. The test cases are usually composed from reported bugs and often have been created by the engineer who fixed the bug or created the feature [27]. Because the concept is aimed for development and continuous integration, it is usually based on a single server that polls version management servers for changes in the source code and, if a change occurred between two polls, compiles the code and performs the test pipeline.

Both concepts have their trade-offs, mainly deriving from their different original projected needs. The keyword-driven test automation is able to test multiple modules with relatively small effort in making the test cases, given that the level of abstract in the test cases is high. When it comes to the project specification, it doesn't matter what kind of control system the keyword-driven concept is testing, since modules are sorted out depending on from which template they were created from. The greatest downside of the keyword-driven testing is that it is only capable of testing single modules at the moment. The framework, for which it is based on, enables more complicated test cases to be built but this hasn't been implemented yet [21]. Features often include multiple modules, as it is easier to test and develop multiple functionalities, in a feature, on their own before integrating them into a feature. This makes it a difficult case for the keyword-

driven testing, in its current state, to validate the integrity of a feature. However, as the keyword-driven concept is aimed for system acceptance testing, it already included a standalone version that could be inserted into a virtualization pool directly.

In data-driven test cases, the test automation has to know which modules and what functionality of these modules is to be tested, which makes it somewhat static since many of the modules are project and product specific. This is the greatest downside of the data-driven method as it only targets modules that have been explicitly defined in a test case, making it dependent on a specific project with predetermined modules. For it to be more dynamic, in terms of project specification, the data-driven test automation needs information about the given specification to generate valid test cases for it.

As a test concept, the data-driven test automation approach was chosen as the most suitable option. The greatest advantage it has over keyword-driven concept is that it supports out-of-the-box functional testing for features that require multiple modules to be utilized and executed in a specific order. This initial readiness enables the preliminary system simulation to be conducted with little effort which makes validating the system concept more practical. The static aspect of the data-driven concept, in terms of the used project specification, was not perceived as a great drawback since it only needs an interface that generates the test cases again, if the project specification differs from which the test cases have been compiled for. The second advantage was the use of open-source based software in the UI, which included numerous plug-ins that could be used to customize the view and execution of test cases as the user perceived best. Also, the system architecture was perceived more intuitive, as one could easily make out from the raw test case files what their purpose was.

The simulation for the data-driven concept was performed in a virtual environment using a ready-made windows-based system that had the MetsoDNA automation platform pre-installed. The purpose was to simulate the workflow of testing a new feature depicted in Figure 13 during the needs analysis phase. The simulation was deemed as simple; accurate enough to prove the communication interface in the test automation was functional and the workflow sequence could be executed. Due to this simplification, no project specification was used, as only one module was generated and uploaded to the PCS. The test case consisted of creating the module, ensuring its integrity, uploading it to the PCS and that a value could be written and read in one of the module's analogy ports.

The simulations of the test concept proved to be successful. The test concept was able to imitate the workflow of testing a new feature with one module. Some preliminary issues did rise in the simulations. For example, the test automation is only capable, at least the current version which was used, of a comparison between absolute values rather than with greater than, or less than, operators. This doesn't create problems for binary signals, but analog measurements, in a step change, follow first order response in control

systems with long settling times, especially if there's some noise in the signal, which make it almost impossible to compare to an absolute value. No UI customization was tested as the preliminary view of the open-source integration platform was considered enough for simple simulations.

After it was validated from the simulations that the concept is functional, and can fill the initial projected need, a project plan for the development of the system was created. Because the simulations were preliminary and issues rose regarding the executing of knowledge worker's workflow, further analysis and design regarding these issues is needed in the form of a prototype. The goal for the prototype is to establish the exact scope of issues that need resolving, their required amount of development and resolving of critical issues which have a large impact on the future development of the system. After resolving the issues in the prototype phase the full-scale engineering design of the system can begin. In the development phases of the system design it is needed to address the third research question: What is the structure of automation system applications and what are their mutual dependencies? This is to create modular, dynamic and robust test automation.

3.4 Engineering design

The chosen concept for system development was the data-driven solution that consisted of an open-source continuous integration web-platform and a test automation tool utilizing Python-language. In the concept definition phase it was stated that the initial simulations were not excessive enough, with issues rising during the test, to validate the commencement of full-scale system design. It was perceived that further analysis of the issues is necessary to validate further development.

The dependence on the initial project specification was identified as the primary issue when generating the modules. As stated above, information about the project specification should descend in the workflow to test cases so that the test automation isn't bound to a specific test project with predetermined modules. If the test automation is bound to a test project, it may seriously restrict the testing of new features in the future because the project modules might lack a functionality needed by the new feature. This will eventually lead to increased upkeep costs, due to the need of manual fixes, as the test automation ages.

The UI issue was the second apparent issue that needed more in-depth analysis. Because all the requirements regarding the UI up to this point have been unquantifiable, no great effort has been made in researching how customizable the open-source application actually is with plug-ins. One of the justifications to choose the data-driven as a preferred concept was the abundance of available plug-ins for the UI, which makes it an important factor to be revised in order to establish a solid base for the justification. Anoth-

er issue that might rise from this is that the plug-ins can make the UI, or even the test automation, unstable and cause conflicts in the software package.

One of the main needs of the test automation was to ease the work of a knowledge worker by automatically verifying the integrity of the feature under development. One of the aspects that rose from the system performance requirements which the planned end-users wanted, was that the test automation would automatically keep the version of SUT up-do-date in terms of the latest changes. The chosen concept offers out-of-the-box support for most of the version control software used in the field of software engineering, but no turnkey solution exists for updating from a local copy. This local copy, as defined in the needs analysis phase, is needed for the knowledge worker to simulate a sandbox environment when developing a new feature. In further analysis it was perceived that creating a separate private branch, in the version control system, for each knowledge worker was deemed as too complex and time consuming to create. This means that another way of fetching the files from the user's local repository must be researched.

The last issue that needed further analyzing was in defining whether a noisy analog signal had reached a threshold. This threshold cannot be addressed with equals-to relative operator, since the noise in an analog signal is not deterministic. To tackle this issue, it is needed to add the relational operators "greater than" and "less than", so that it is possible to approximate the measured values. The lack of these relational operators was seen as a small aspect, which should still be analyzed in terms of work hours. This is to ensure that the relational operators can be added without extensive work related in modifying the test automation source code which might pose a development risk in terms of schedule.

3.4.1 Prototype

After analyzing all the issues listed in the previous paragraphs, it was perceived that more in-depth simulation was considered necessary to resolve these issues, or at least scope the amount of work needed. The simulation was decided to be executed in the form of a fully functional prototype. The goal of the prototype is to mitigate development risks from identified issues by resolving them with preliminary system design.

The first design issue in regarding the prototype was to create a proper UI to manage the test cases. It was analyzed that the test cases should be maintained from programs that offer table-tools, such as Microsoft Excel, as they provide more user-friendly environment for editing than plain text files. Each test case has its own file which depicts the sequence, steps and actions the test case must perform.

To resolve the issue with project specification dependence, it was necessary to supply the test cases with project information so that they could be linked to the correct mod-

ules. To achieve this, a script was created that compiles the test cases using project information before every test run. The workflow of compiling the test cases is done by taking the test case's template file and transforming it into a format that the test automation application can read, in this case the Extensible Markup Language (XML), and setting the module names as specified by the specification. It was not yet possible to create a direct link between the creation of project specification and compiling of test cases but this is a feature that is to be implemented later on, not in the scope of this thesis, which has to be taken into account when creating the script. Because of this the script was made as dynamic as possible. As input, the script takes the information about the generated system and fills in the ports, that the test cases utilize in the functional testing, to match the generated modules. An intelligent script that only compiles test cases, to which the changes in the specification are directed, was not perceived as necessary since the compiling of all test cases is deemed simple and fast to execute with modern processors, even when using a large test case library.

To support the future direct link from project specification, all the dynamic port names in the test case template files were named so that they could be easily detected and replaced with the information descending from the specification file without sacrificing the integrity of the files, for example "%%Measurement1%%". This whole workflow is depicted in Figure 14. The direct link from project specification creation is marked with a dashed line.

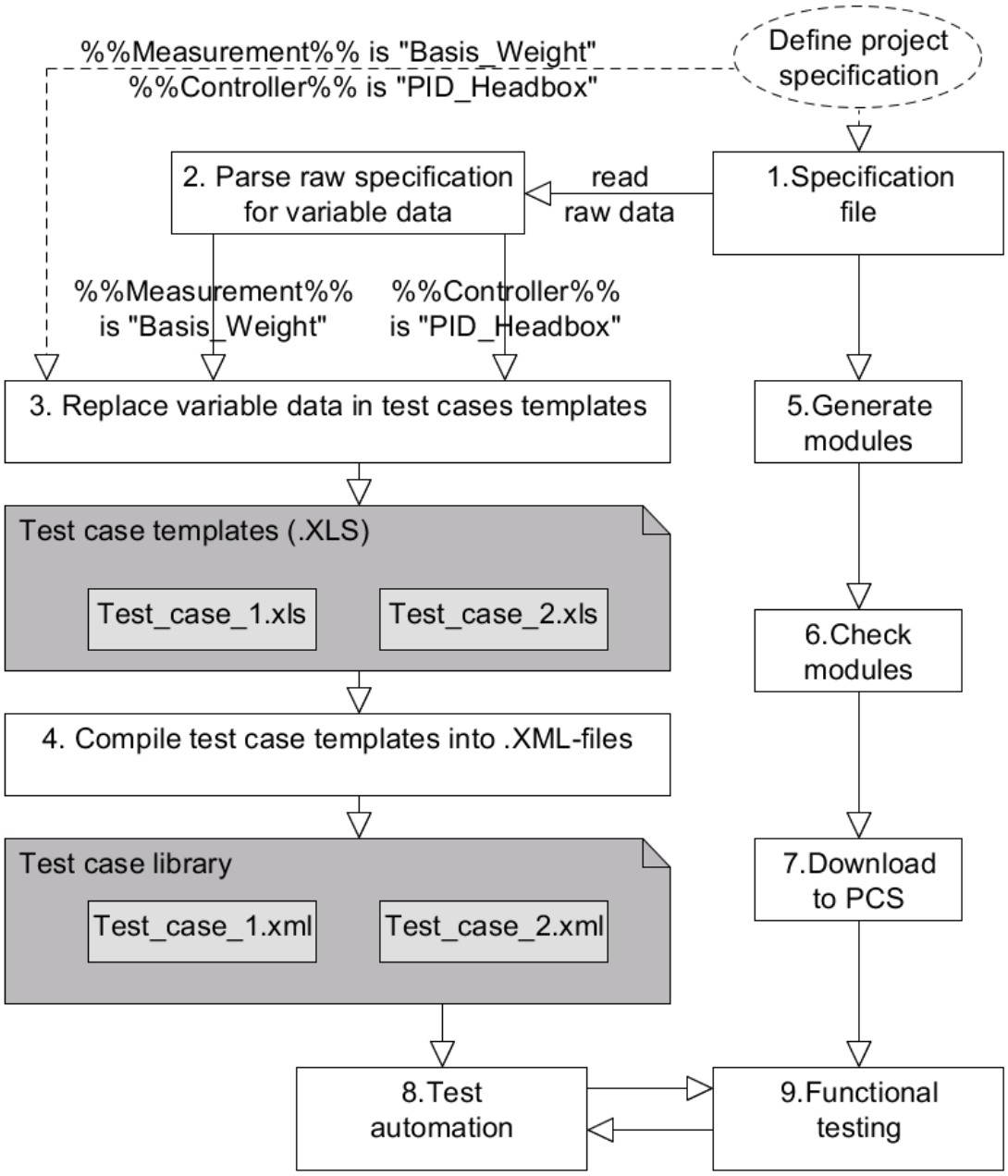

Figure 14. Workflow of compiling a test case.

The issues with defining an agreeable UI was resolved with researching the numerous plug-ins provided for the open source application platform. From the system performance requirements it was derived that the production chain should be visible and easily operated from one single view. From the numerous alternatives researched, a plug-in was taken into further analysis which provides a view of connected actions that typically form a deployment pipeline. Deployment pipeline is a predetermined set of validation actions through which the software, or a part of it, must pass before it can be released. The preliminary view of the pipeline is depicted in Appendix A. The colors and the states they represent are shown in Table 1.

Table 1. Deployment pipeline colors and their states.

Color	State
Green	A successful run without any errors or warnings.
Yellow	A build is ongoing and the progress is represented in the loading bar.
Orange	The build was completed but was unstable. A run can be unstable if any warnings arose during the action.
Red	The build has failed. This is because of an error in one of the test case steps.
Blue	The build hasn't started yet.
Gray	The build was aborted by the user

It should be noted that in this prototype version the pipeline continued even though an action failed, which is not a valid situation in the final concept. The reason it was configured as such was to gather information from each step run, were they successful or not, to debug arising issues during the building of the prototype.

Creating a polling mechanism to keep the test automation up-to-date from a local copy was identified as an issue to be solved, as it's considered as an extension to the already implemented pipeline plug-in. In the research multiple plug-ins, from here on build triggers, that provided this kind of functionality were found, with slight differences in functionality. Simply put, there were two kinds of build triggers for monitoring the file system. The first build trigger triggers a build when certain files are found in a specific directory. This could be implemented in such a way that each time new files with the added feature are copied to a specific folder in the test automation server, the pipeline initiates.

The second build trigger triggers a build when a file, or a set of files, has changed. This option would require access the user's hard-drive and the build trigger configured to poll the development folder remotely for changes. From these two options the latter one was seen as more preferable since, if configured correctly, the user only needs to save the changes done into local repository, which will initiate the build pipeline. It was perceived that there needed to be a time-out period which the build trigger will wait before initiating the build pipeline, since it takes time for the user to save the changes done to multiple modules in the local repository. The build trigger was configured to poll the local repository of a simulated user over the network. The pipeline with the added build trigger plug-in is depicted in Appendix B. The color codes are the same as in stated in Table 1.

However, it should be noted that this second build trigger plug-in uses Message-Digest algorithm 5 (MD5) to define whether the contents of a folder have changed or not, which is known to have collision vulnerabilities from as early as 1996 [30]. Collision vulnerability means that with two different files, the algorithm produces the same hash value for both. Hash value is used to define whether the files are identical, which would give a false positive in the case of collision. This wasn't seen as a great risk in this application, as the only practical problem that could arise from this is that the build pipeline doesn't initiate even though the user has done changes to the local repository.

When identifying the needs for a knowledge worker's and for the whole product scope testing, it became evident that there needed to be two separately configured versions of test automation. The first one is a build trigger linked to the knowledge worker's sandbox environment. The second one is a build trigger coupled with version management software that enables the testing of the latest stable versions in the entire product scope. The latter one, with the link to version control software was not implemented in the prototype, as the functionality has already been validated and implemented in the original application, from which this test automation concept was derived from [27].

The issue with relative operators during testing was researched in co-operation with knowledge workers who had initially developed the test automation application. It turned out that the relational operators are a standard part of the programming language and could easily be initialized during a test step with a specific parameter. To ensure the expression work as intended, a functional test utilizing relational operators was created.

To simulate the functional testing part of the prototype, a simple test case to achieve this was constructed. This test case turns on the simulation on by checking first that the values in the module are at defaults, then initiating the simulation by writing binary values, checking that the simulated measurement reaches the predetermined value within a set time limit and finally restores all the modified values back to default and waits for a while to allow the values to settle down. Because the order of the measurement responses are first degree, it was necessary to compare the value with greater-than relative operator, since comparing absolute values with equals-to operator can yield false positive results due to numerical inaccuracy.

The results from the testing of the prototype were deemed as successful and the preliminary system addressed all the major issues that rose in the analysis phase. The system was functional and already fulfilled the preliminary need set by the knowledge workers. No major issues rose during the testing of the prototype, but it was deemed that the UI for creating the test cases needed to be improved later on in the engineering phase.

3.4.2 Design phase

The design phase was started by analyzing the system performance requirements that were not fulfilled in the previous prototype phase. The first unattended issue was the reporting of each test phase, which had to be taken into account as a primary issue since it's closely bound to the design of UI, in terms of how the reports are presented and how is the user alerted when issues during testing rise.

The test automation in its current state, following the prototype stage, provides a thorough report while executing a test phase and after the run has completed. This report, referred as a log file from here on, was perceived as too detailed for a knowledge worker to examine as it contained all the actions the test automation performs which are only useful during the development of the test automation. An example of a log file generated is depicted Appendix C.

To highlight the most important lines from the log file, a plug-in was used that provides a customizable failure library, to which the users can add the causes of build failures during testing. This build failure analyzer automatically scans every log-file and highlights if it finds the line specified by the user. The lines are searched by the means of regular expression. Regular expression can be defined as an expression which describes a set of strings or a set of ordered pairs of strings [31].

To further improve the availability of the log file, the test automation was configured to send an email for every failed or unstable build. This option was integrated to the webserver software, but it was further extended with a plug-in to gain freedom in modifying the sent email message. General info about the build and the set of changes were added to the message, as they were perceived to be useful in case the build failed. The build failure analyzer was also integrated with the email plug-in, so that every email sent by the test automation included the indicated fault and also direct link to the fault detected that highlights the line in which the fault was detected. An example email is presented in Figure 15.

Figure 15. An example email notification sent by the test automation in case of build failure.

To maximize the success of evaluation and testing the system in terms of test cases, interactions and characteristics between the applications in the entire product scope were researched. This will greatly benefit the creation of test cases, as the interactions between separate products are known, and dealing with the issue of change management. Change management in this case means constructing the test automation so that it can be exported to test other automation applications, not currently present in this application scope, with ease. The method used to research interactions and characteristics was graph theory in which digraphs were utilized to visualize the connections.

The research of characteristics was started by generating an example system which included as many automation applications as possible. As a starting point, an old customer case was used to minimize the errors rising in creating the project specification, because the creation of an entire project specification from scratch can be perceived as a feat of its own that would require great effort in term of work hours.

The old customer case is a two-layer cardboard machine with a wide set of control products, including MD control system. The product scope of the cardboard machine was extended with as many products as possible, but still trying to keep the system set-up realistic. This means that mutually exclusive products and measurements that are not typical in this type of cardboard machine were not included. Some issues rose in adding new products to the initial project specification since some parts of the project specification were outdated, but this was fixed by updating all products to the latest version with the help of knowledge workers.

When all the automation modules for the entire project specification had been created, they were sorted and named depending on the automation application from which they were created from. To analyze the interconnections between the automation modules, a tool already present in the R&D was used to compare the input and output ports and to determine which of them were interconnected. This tool analyzes the metadata from the automation modules and collects information about input and output ports but also the product version from which it was generated from. Some small modifications had to be done to the tool in order to analyze all the automation modules present, since it was originally aimed for comparison between a few products.

An example of the high-level product map of this tool is presented in Figure 16. This product map depicts the main product families where each sub-product belongs to. A more in-depth product map with sub-product interconnections was created. The sub-product map consisted of 64 vertices and 381 arcs, and because of its size and complexity it cannot be depicted as a figure in thesis.

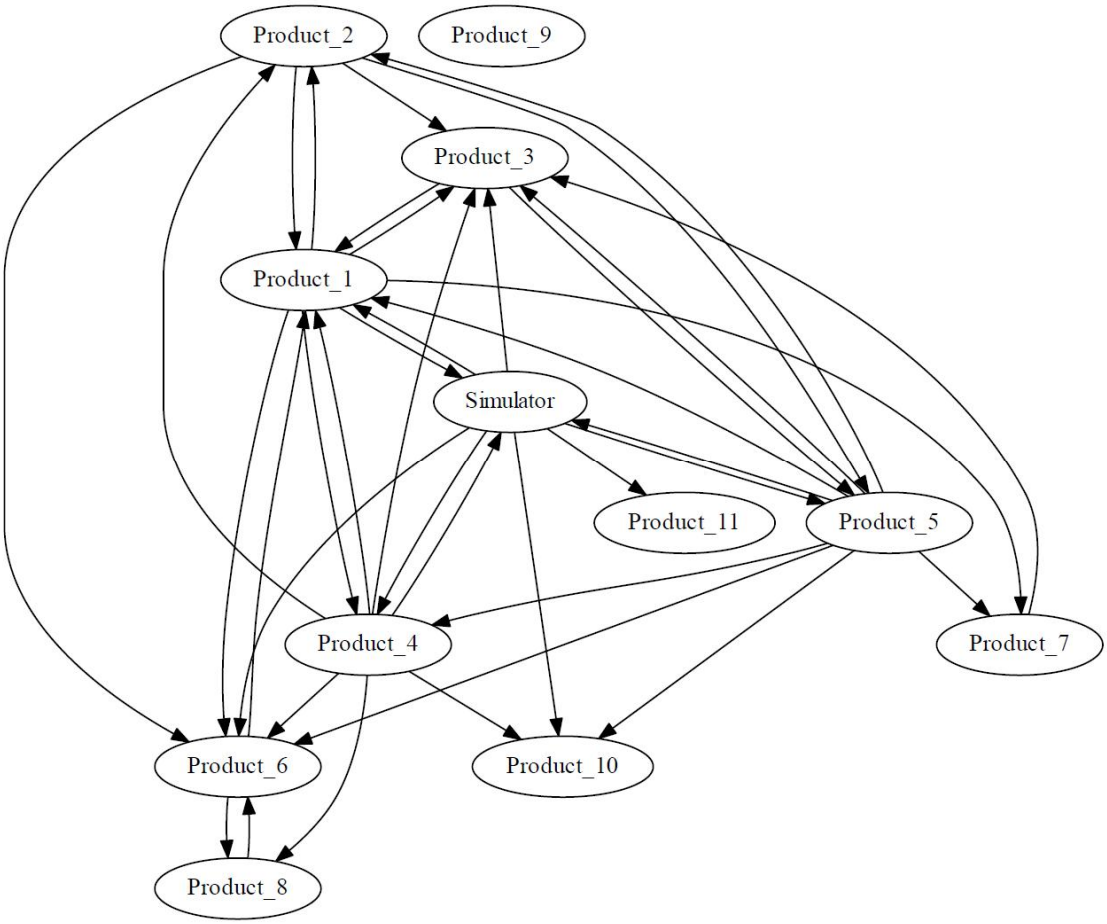

Figure 16. Digraph depiction of a product map.

The analysis produced interesting results, such as situations where a product family didn't seem to have any initial connections to other product groups. An example of this is the "Product_9" depicted in Figure 16. There are a couple of reasons for such an event. One is that some of the connections between the products are done on the hardware levels, which are not explicitly specified in the automation modules that provide the functionality. Another reason is that even though the generation of the automation modules has been proved to be functional, small errors in project specification can cause some ports to be generated with an invalid tag.

To mitigate the maintenance risk in terms of project specification, a test-case compiler was to be designed. The scripts from the prototype phase were used as a valid starting point, since they had the preliminary functionality needed. The prototype version were designed for simple systems, and needed to be extended to cover multiple products. The script was extended so that with one template test case, it was possible to generate all the controller-measurement pairs. The multiple controller-measurement pairs were simply duplicated from the original template test case with correct port names and added to the end of the previous test step in the test case file.

However, an issue rose in creating test cases for large, complex system. In more specific test cases, where the functionality of a separate product scope was tested, the ports used were strongly product specific. The interconnections between the products can be seen from the product analysis depicted in Figure 16. In further design analysis it was perceived that the test case templates needed more parameters to be practical.

The first parameter needed by the test case templates was the specified product, to define the initial focus for the test case. This is needed because the port names are dynamic by design and depend heavily on the product scope selected and thus cannot be multiplied for all product families. This means that if the project has multiple actuators, measurements and control products, not all the ports should be generated since some of them are product specific. For example, in a three layer cardboard machine only the test cases for actuator-measurement pairs should be generated, since it's impossible to control the quality measurement of the middle layer with upper layer's actuator. For general ports, such as in simulation and on/off-switches, the parameter can be set to "Any", since they are created for all products in the same fashion.

3.4.3 Implementation

The implementation was conducted in a virtual environment which had an operating system and the newest version of automation system platform pre-installed. To ensure the test automation had enough space to operate, and that it could be ported to other workstations with relative ease, a dedicated hard-drive was allocated for the test automation files.

Implementation started with installing all the runtime environments needed by the web server and test automation core programs. Configuration was done by hand and the test automation was set to poll another virtual machine to simulate the sandbox environment requirement. The workflow configuration was copied straight from the prototype and it needed only minor configurations to work in a new environment, since the scripts were created to be as dynamic as possible in terms of runtime environment.

The installation and configuration of the whole test automation program was done without any major issues. It was perceived during the implementation that a setup program, which could automatically set all the configurations, would ease the process of setting up the system but was seen as too much work considering the test automation is only preliminary.

3.5 Testing and system validation

The functional testing of the implemented test automation was initiated by creating a master test plan. The master test plan consisted of system and integration testing. Unit testing was perceived as unnecessary since it had been initially done in the previous

prototype phase. The test plan consists of functional tests only, which are aimed to address the most important functionalities. Non-functional characteristics, such as usability and security, were not tested.

The main objective of the master test plan is to validate that the test automation can continuously execute the steps of the workflow, depicted in Figure 14, and that the results from the workflow are consistent throughout the tests with no unexpected errors rising. The main functionalities that must be validated in order to achieve the goal are: The test cases are generated correctly using the project information, test automation automatically starts when a change occurs in a remote location and detailed report from each workflow phase is sent to the user with comprehensive information about the current state of the test run.

Achieving the validation of the said functionalities was done by simulating the sandbox environment in which the development workers create new features. First, the test automation was configured to poll another virtual machine's local folder for changes to include the trigger for automated builds. For testing the consistency of the system's workflow, including the compiling of test cases, the setup was configured as an infinite loop. The configuring was done so that when the last functional test case had finished executing, a change will occur in the remote sandbox environment, which the test automation is constantly polling, causing the workflow to start again. The change was done by simply renaming a dummy test file, which had no embedded functionality, since the polling mechanism was set to monitor only the state of a folder. Despite the phases of the workflow being successful or not, an email report was configured to be sent to validate the reporting aspect. It should be noted that for testing purposes, the system had to be reset first for each run by deleting all modules present in the system and clearing any configuration files. Also, the generation and importing of modules was combined into one single step, due to the fact how the module generation tool was implemented.

The whole workflow was run a total of 50 times and from each run the build durations and states were recorded to a log files from the web server for further analysis. The amount of emails received throughout the test period was checked against the number of workflows executed to ensure all the phases were accounted for.

4. RESULTS

The V-model provided a functional preliminary test automation system that met the initial projected need by the R&D department. The system development process produced a virtualized test automation which can be duplicated and, with minor configurations, initialized for each development worker to provide their own sandbox-environment.

The test application fully automates the workflow of a feature test previously done by the knowledge worker, which directly saves work hours. The work hours saved can now be used to improve and fix possible features during the development process. However, it should be noted that with the test automation, the development worker should create test cases during the development of a new feature, so that they can be added to the global test case library which is used to test the whole product scope. As a result, the creation of test cases was made as easy as possible from the development worker's point of view, so that it doesn't mitigate the work hours saved by the automated workflow.

From the 50 workflows ran during the testing phase, mean and two standard tions 2σ, also known as 2-sigma, were calculated for all phases. Standard deviation is the square root of variance. Variance represents the average of the sum of squared deviation scores. Deviation score is the numerical distance between the measured value and the measured data set's mean value. In normal distribution, the standard deviation depicts the probability that an observation value lies within its limits, which for one standard deviation σ is 68.25 % and for two standard deviations 2σ is 95.44 %. This characteristic clearly depicts how much does the observation value fluctuate in the test runs. [32, pp. 259-260]. The average run times of each workflow phase, as seen in Figure 13, are presented below in Table 2.

Table 2. Average run times and 2-sigmas for development workflow phases.

Phase	Mean (seconds)	2-sigma (2σ)
System reset	702	28
Generation	669	114
Check	867	135
Download	602	45
Functional testing	240	3
Total duration	3080	325

5. CONCLUSIONS

The systems engineer method V-model was successfully used to develop preliminary test automation for paper machine's MD control system. The development started with analyzing the needs for the test automation application in the R&D department. There was no predecessor system present that could automatically test the MD control system, which made the research needs-driven. It came apparent in the early stages of the research that the lack of industry-wide standard for automation application development will make the use of commercial third-party test automation difficult. The needs analysis revealed a workflow, which the developer has to go through in order to start the functional testing of a new product or feature. This was identified as an important and valid starting point to begin developing test automation. From this workflow, a set of operational requirements for the test automation were aggregated in co-operation with knowledge workers. Operational requirements describe the purpose of the system, its capabilities and how it is deployed. Numerous use-cases were refined from the operational requirements that depict what scenarios the planned test automation should be able to perform during its life cycle.

Because the architecture of an automation system can vary greatly between manufacturers, it was perceived necessary to explore for possible test automation concepts from inside the R&D department. To achieve this, a set of unstructured interviews were held with development workers who have done research regarding test automation in the R&D department. The interviews proved to be useful, as numerous different approaches to develop test automation for automation systems had been researched in the past. Two strong candidates rose as the most promising options for further development, keyword-based and data-driven test automation. With the explored system concepts, it was possible to produce system performance requirements that deliver more in-depth requirements for the projected test automation.

From the explored test automation concepts, a data-driven method was deemed as the most suitable option in this particular case. The data-driven method contains a test case library which consists of a number of individual test cases which the user can run in arbitrary order. This test concept is aimed for testing a specific functionality or aspect of a feature which made it a superior choice.

Even though the data-driven concept was chosen as the most appealing test concept, it should be noted that the keyword-based testing system is superior when it comes to test-

ing a great number of modules. In the span of this thesis, it was not possible to combine the pros of both test concepts, but should be considered in future work. The web-server based user-interface has the functionality to monitor remote tasks, which means the template-based testing concept could test a part of the deployed system together with functional testing.

An initial prototype of the chosen data-driven test concept was constructed, which addressed all the identified issues that had risen in the research. The issues ranged from test case management, project specification management to workflow initiation triggers. The prototype successfully resolved all the issues identified with no new major issues rising during the testing and evaluation.

The design phase of the test concept consisted of creating satisfactory reports from the test automation to the user and to increase user-debugging performance by improving the readability of the log-file. To maximize the success of evaluation the system in terms of test cases, the interactions and characteristics between the applications in the whole product group were researched using graph theory as a method.

The result from the research of characteristics between sub-products in a product group provided a product map with 64 vertices and 381 arcs. The product map is very useful in future development of the test automation, even if the presented concept isn't utilized, since it gives a good basis from which the different products should be tested. If, for example, each sub-product has a two-minute-long test case and none of these test cases can be run in parallel, the total testing of all sub-products will take 128 minutes. Of course, when a developer does a minor change in a sub-product, the modified set of products should get tested first. But which products should be tested afterwards to optimize the use of resources while keeping the total test time reasonable?

The analysis of the characteristics between products provides a solid basis for determining the priority of the test cases. Since the analysis provided digraphs, that indicate which the direction of the information between products, the test cases can be prioritized as which ones have the most outputs to other products. Another way to further develop the prioritizing is to find which products take inputs from the modified product. This is based on the idea that if all the unmodified products are error-free, they can only be broken by invalid inputs from the modified product. With this basis, all the products which take inputs from the modified products should be tested as a high priority. The priority chain can be expanded to apply products which take inputs from products that have a high priority, due to being linked directly to the modified product.

In the evaluation phase, it was proved that the developed automation system successfully decreases the time a knowledge worker has to invest in testing during the development of an automation application. Even though the functional testing phase in the

workflow was tested with only one simple measurement test, it was enough to verify that the functional test cases work as intended.

In the future, the test automation should be improved by adding valid test cases which are generated during the development of a new feature. By adding more test cases to the test case library the test automation can catch more errors and save more work hours. In addition to the test cases added, a specific fault which can rise from the new feature should also be added to the provided build failure library. This helps to pinpoint the errors rising from new products.

The usability for change management regarding the test cases should also be researched more in the future. The constructed preliminary tool for creating test cases was only to prove the functionality of the concept and didn't contain any scripts that could ease the work a developer has to do to create a test case. One approach for this would be to make a wizard-type creation for the test cases such that the only thing the user needs to input is the port's name and then choose what action is executed with it. This would mitigate the errors rising from incorrect format and typos that cause problems in the preliminary version of the test case template.

For the test cases, there should be less static variables included. This is the case, for example, in comparing analogue values from a measurement. If they are defined just as scalars in the test cases, then the specific case is bound to one project specification with predetermined minimum and maximum values for an actuator. To tackle this issue, there should be an option for simple calculations in the values that are descended from the project specification. Also, if possible, the value could be formed as a sum from other ports which means the test value changes according to the operating point of the process.

The proposed features in the previous three paragraphs would reduce the amount of change management when the test automation ages and, if designed carefully, bring new value for the test automation as it learns new defects via build failure library and test case creation wizard. Even though the deployed system was a preliminary version, the added value from the proposed features prevents the test automation from becoming obsolete ahead of time, keeping the chosen test concept as a viable option for future development.

REFERENCES

[1] D. M. Harland, "The management of software engineering," *IBM Systems Journal*, vol. 19, no. 4, pp. 414-420, 1980.

[2] A. P. Sage ja W. B. Rouse, Handbook of systems engineering and management, Hoboken, New Jersey: John Wiley & Sons, 2011, p. 1504.

[3] A. Kossiakoff, W. N. Sweet, S. J. Seymour and S. M. Biemer, Systems engineering: Principles and practice, Hoboken, New Jersey: John Wiley & Sons, Inc., 2003, p. 528.

[4] Agile Finland, "Manifesto for Agile Software Development," 1 February 2001. [Online]. Available: http://agilemanifesto.org/iso/en/. [Accessed 8 December 2014].

[5] J. Ollanketo, *Project Manager*, Tampere: Metso Automation Oy, 2014.

[6] W. W. Royce, "Managing the Development of Large Software Systems," in *IEEE WESCON*, Los Angeles, 1970.

[7] B. W. Boehm, "Software Engineering," *IEEE Transactions on Computers*, Vols. C-25, no. 12, pp. 1226-1241, 1976.

[8] S. McConnell, Code Complete: a practical handbook of software construction, Redmond, Washington: Microsoft Press, 1993, p. 857.

[9] D. M. Buede, The Engineering Design of Systems: Models and Methods, New York: John Wiley & Sons, 2011, p. 536.

[10] A. P. Sage, Systems Engineering, New York: John Wiley & Sons, 1992, p. 606.

[11] G. Hassan, Software modeling and design: UML, use cases, patterns, and software architectures, New York: Cambridge University Press, 2011, p. 550.

[12] Suomen automaatioseura, Automaatiosovellusten ohjelmistokehitys: Suunnittelun työtavat, välineet ja sovellusarkkitehtuurit, Helsinki: Suomen automaatioseura, 2005, p. 151.

[13] S. H. Kan, Metrics and Models in Software Quality Engineering, Boston, MA: Addison-Wesley Longman Publishing Co., 2003, p. 528.

[14] P. N. Ghauri ja G. Kjell, Research Methods in Business Studies: A Practical Guide, New York: Pearson Education, 2005, p. 257.

[15] M. A. McDaniel, D. L. Whetzel, F. L. Schmidt and S. D. Maurer, "The Validity of Employment Interviews: A Comprehensive Review and Meta-Analysis," *Journal of Applied Psychology*, vol. 79, no. 4, pp. 599-616, 1994.

[16] R. L. Dipboye, "Structured and unstructured selection interviews: Beyond the job-fit model," in *Personnel and Human Resources Management*, vol. 12, Greenwich, JAI Press Inc., 1994, pp. 79-123.

[17] T. May, Social Research: Issues, Methods and Research, Berkshire: McGraw-Hill International, 2011, p. 354.

[18] K. Ruhonen, ”Graafiteoria,” Tampere University of Technology, Tampere, 2014.

[19] J. Bang-Jensen and G. Z. Gutin, Digraphs: Theory, Algorithms and Applications, New York: Springer Publishing Company, 2008, p. 798.

[20] "System platform: Metso.com," Metso Automation Oy, 2014. [Online]. Available: http://www.metso.com/Automation/process_prod.nsf/WebWID/WTB-120829-22575-4834B?OpenDocument. [Accessed 11 August 2014].

[21] M. Karaila, Interviewee, *Research Manager*. [Interview]. 26 May 2014.

[22] P. Kotiluoto, Interviewee, *Program Manager*. [Interview]. 24 June 2014.

[23] PLCopen, "PLCopen: IEC 61131 Standards," PLCopen, 3 September 2013. [Online]. Available: http://www.plcopen.org/pages/tc1_standards/. [Accessed 7 August 2014].

[24] O. Sharma, "Cross Browser Incompatibility: Reasons and Solutions," *International Journal of Software Engineering & Applications*, vol. 2, no. 3, pp. 66-77, July 2011.

[25] Awio Web Service LLC, "WC3Counter: Global Web Stats - July 2014," Awio Web Service LLC, 31 July 2014. [Online]. Available: http://www.w3counter.com/globalstats.php?year=2014&month=7. [Accessed 4 August 2014].

[26] C. Nagle, "Test Automation Frameworks," SourceForge, [Online]. Available: http://safsdev.sourceforge.net/DataDrivenTestAutomationFrameworks.htm. [Accessed 26 August 2014].

[27] S. Virtanen, Interviewee, *Project Manager*. [Interview]. 27 May 2014.

[28] S. McLaughlin and P. McDaniel, "SABOT: specification-based payload generation for programmable logic controllers," in *ACM conference on Computer and communications security*, New York, 2012.

[29] A. Jääskeläinen, *Design, Implementation and Use of a Test Model Library for GUI Testing of Smartphone Application*, Tampere: Tampere University of Technology, 2011, p. 68.

[30] H. Dobbertin, "The Status of MD5 After a Recent Attack," *RSA Laboratories CryptoBytes*, vol. 2, no. 2, p. 16, 1996.

[31] R. Mitkov, The Oxford Handbook of Computational Linguistics, New York: Oxford University Press, 2005, p. 786.

[32] O. Ibe, Fundamentals of Applied Probability and Random Processes, Burlington: Elsevier Science, 2014, p. 457.

APPENDIX A:

Figure 17. Deployment pipeline plug-in.

APPENDIX B:

Figure 18. Deployment pipeline plug-in with build trigger.

APPENDIX C:

```
2014-09-13_17.10.50       Success
2014-09-13_17.10.50       messages:
2014-09-13_17.10.50     1 PostConditions
2014-09-13_17.10.50       ?AppValueExpression
2014-09-13_17.10.50         user variables:
2014-09-13_17.10.50          _refVal (_refVal): [<System.Byte
object at 0x000000000000003A [0]>, 5000.0]
2014-09-13_17.10.50          _default (Module1:port1):
[<System.Byte object at 0x000000000000003B [0]>, 5000.0]
2014-09-13_17.10.50         expression: _default.member
("").value == _refVal.value
2014-09-13_17.10.50         Module1:port1 == [0,5000.0] is
True ( value is [<System.Byte object at 0x000000000000003C
[0]>, 5000.0] )
2014-09-13_17.10.50         Success
2014-09-13_17.10.50     step ok
2014-09-13_17.10.50 +pass+ test 44: Enable simulation and
check that it reaches the default measurement value in
specified time <7030_Simulation_ON>
2014-09-13_17.10.50 (NoExecMode) Total tests ..... :
48
2014-09-13_17.10.50 (NoExecMode) Skipped tests ..... :
47 (filter was active)
2014-09-13_17.10.50 (NoExecMode) Passed tests ..... :
1
Finished: SUCCESS
```

Figure 19. Example log-file.